Robustness of Variable Length Codes

Stephen James Spackman

Division of Mathematics and Statistics
University of Glamorgan/Prifysgol Morgannwg

Master of Philosophy

January 2001

R11

Certificate of Research

This is to certify that, except where specific reference is made, the work described in this thesis is the result of the candidate. Neither this thesis, nor any part of it, has been presented, or is currently submitted, in candidature for any degree at any other University.

Signed ..... S.S pachuan .....
Candidate

Signed ..... Perkins .....
Director of Studies

Date ..... 23.1.01 .....

Abstract

The aim of this thesis is to determine the robustness of certain variable length codes in the presence of channel errors. The codes studied are: Huffman, Huffman equivalent, and T–codes. The algorithm used to construct each code is presented, together with appropriate methods of modifying the constructed code. The synchronization properties of such codes are discussed.

The robustness of each code is measured and compared by considering its average error recovery after an error has occured. Two theoretical models, based on Markov chains, are presented and contrasted and the most accurate one selected. A simulation of the codes has also been implemented, and observed error recovery averages recorded. Both the theoretical and practical observations are used to draw the necessary conclusions.

This work will demonstrate that Huffman equivalent codes are more robust than modified T–codes and Huffman codes. It will also be observed, that unmodified T–codes are competitive with Huffman equivalent codes. However, it is believed that unmodified T–codes are impractical as such codes are hard to construct or indeed nonexistent for many distributions.

Other observations will emphasize the accuracy of the chosen Markov model, and the resilience and stability of robust codes in a noisy channel. The theory will be shown to be consistent with the simulation and thus validated.

Acknowledgements

I would like to thank my supervisors Dr. Stephanie Perkins and Prof. Derek Smith for their guidance in this thesis. I would also like to thank the University of Glamorgan for supplying some funding which allowed me to complete this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction.

1.1 Information Theory and Redundancy.

Information theory is a branch of mathematics concerned with efficient storage and transmission of information. Data used to represent information can contain an inherent amount of redundancy. Thus redundancy will affect the efficiency of data storage and transmission.

1.1.1 Information Storage and Transmission

Information can be represented as an arrangement of source symbols termed data. To store or transmit data the medium used is termed a channel. A channel has an alphabet associated with it. Only symbols in this alphabet can be sent across it. The method of converting data into a form that is suitable for the channel is termed encoding. The encoded source symbols are referred to as codewords and a set of codewords is called a code. Following the storage or transmission of encoded data, a process of decoding is applied to the codewords to retrieve data. However in practice the data retrieved may not equal the original data sent, due to the presence of channel errors corrupting the encoded data.

The process of information representation, transmission and storage in the presence of channel errors is expressed pictorially in Figure 1.1: with "SOURCE" and "SINK" referring to the information represented prior to encoding and after decoding, respectively.

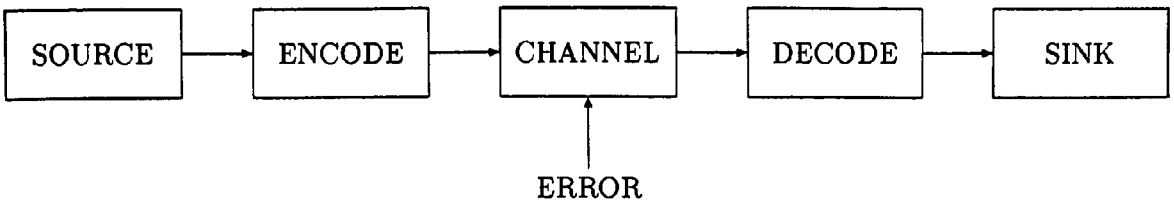


Figure 1.1: Information storage and transmission model.

An example of a code used for data storage and data transmission is the ASCII (American Standard Code for Information Interchange) code [7]. This code is composed of 128 codewords, each one representing a symbol or an instruction. For example, the ASCII codewords representing the lower case letters of the English alphabet are given in Table 1.1; with each codeword composed of symbols from the set $\{0, 1\}$.

Symbol	Codeword	Symbol	Codeword
a	10000010	n	10011100
b	10000100	o	10011111
c	10000111	p	10100000
d	10001000	q	10100011
e	10001011	r	10100101
f	10001101	s	10100110
g	10001110	t	10101001
h	10010000	u	10101010
i	10010011	v	10101100
j	10010101	w	10101111
k	10010110	x	10110001
l	10011001	y	10110010
m	10011010	z	10110100

Table 1.1: The ASCII code.

Writing codewords in terms of elements from the set $\{0, 1\}$ allows for practical implementation of codes. For example, the symbols '1' and '0' may be represented by an electric current or no electric current, respectively. Hence information storage and transmission is realizable.

1.1.2 Channel Errors and Robustness.

Encoded data may be subject to corruption during transmission or storage. The corruption occurs when coded data is subjected to random channel errors. For codewords constructed from the set $\{0, 1\}$ there are three types of random errors: bit inversion; bit loss and bit gain:

- Bit Inversion: a bit inversion error is one that changes a bit into another. For example, a '0' becomes a '1', or a '1' becomes a '0'. A combination of bit inversion errors can also occur to a codeword. For example: '0110' might become '1100'.
- Bit Loss: if a bit loss error occurs in a codeword, the result is a reduction in the codeword length; due to the loss of at least one bit. For example: '0001' becomes '000'.
- Bit Gain: similar to a bit loss error, except that a bit is gained. For example: '0011' becomes '01011'.

Channel error is a practical problem. The effect of errors on coded data varies: from significant corruption, causing loss of information, to minimal corruption that is negligible. If a code is to be implemented, the tolerance of the code to channel errors must be observed: a code that loses significant amounts of information is not practical, compared to a code that is error tolerant. Hence for practical purposes the robustness of a code must be researched.

1.2 Variable Length Codes and Data Compression.

1.2.1 Variable Length Codes.

In a block code all codewords are of the same finite length. The ASCII code is an example of a block code. A code composed of codewords not all equal in length is termed a *variable length* code [7]. The Morse code [7] is an example of a variable length code that can be used for encoding letters in the English alphabet. In Table 1.2 each Morse codeword is composed of one or more bits.

However in practice the Morse code is not a binary code. To identify each coded letter, a space is required, and to identify each coded word at least two spaces are required. To represent the space character a third symbol is used: '2', for instance. Thus the Morse code is in fact a *ternary* code, one requiring symbols from the set $\{0, 1, 2\}$ to encode and transmit data.

Symbol	Codeword	Symbol	Codeword
a	01	n	10
b	1000	o	111
c	1010	p	0110
d	100	q	1101
e	0	r	010
f	0010	s	000
g	110	t	1
h	0000	u	001
i	00	v	0001
j	0111	w	011
k	101	x	1001
l	0100	y	1011
m	11	z	1100

Table 1.2: The Morse code.

The Morse code is based on a property inherent in English language text: it is observed that some letters occur more frequently than others. In a large volume of text, the letter "t" is expected to occur more frequently than the letter "z", thus "t" occurs with greater probability than "z" for example. Taken from [5], the probability of each letter in the English alphabet is listed in Table 1.3.

The Morse code reflects the letter frequencies by assigning to each letter a codeword of variable length: the more often a letter occurs, the shorter the codeword length, and the less often a symbol occurs, the longer the codeword length.

1.2.2 Data Compression.

Data compression is the process of reducing the amount of data required to represent a given quantity of information [11]. For example, a consequence of using a Morse code rather than the ASCII code is the comparative reduction in the number of symbols used to encode the data. Encode the word "attack" using the ASCII code by substituting each letter with the codeword equivalent:

Letter	Probability	Letter	Probability
e	0.1278	m	0.0288
t	0.0855	p	0.0223
o	0.0804	f	0.0197
a	0.0778	y	0.0196
n	0.0686	w	0.0176
i	0.0667	g	0.0174
r	0.0651	b	0.0141
s	0.0622	v	0.0112
h	0.0595	k	0.0074
d	0.0404	j	0.0051
l	0.0372	x	0.0027
u	0.0308	z	0.0017
c	0.0296	q	0.0008

Table 1.3: Letter frequencies in the English language.

Data	Encoded Data
attack	10000010 10101001 10101001 10000010 10000111 10010110

and similarly encode the same information with the Morse code by substituting each letter with the equivalent Morse codeword:

Data	Encoded Data
attack	01 1 1 01 1010 101

It can be seen that less data is used to represent the same information, and evidently less redundancy was present in the Morse encoding.

There are practical consequences to implementing data compression techniques for information storage and transmission. The cost of storage space is expensive, using compressed data to store information can reduce this cost. Transmission of data from one source to another can be time consuming if the information transmitted is lengthy. Transmission of compressed data reduces the time taken

for information transmission.

1.3 The Average Codeword Length and Entropy.

The average codeword length of a variable length code C is

$$\sum_{c \in C} (P(c) \times \mid c \mid) \text{ bits}$$

where c has probability $P(c)$ and length $\mid c \mid$ bits. For example, the average codeword length of the variable length code $C_{example}$

$Probability$	$C_{example}$
0.4	00
0.2	01
0.2	10
0.1	110
0.1	111

is

$$0.4 \times 2 + 0.2 \times 2 + 0.2 \times 2 + 0.1 \times 3 + 0.1 \times 3 = 2.2 \text{ bits.}$$

The average codeword length of a variable length code C measures the average number of bits required to represent each source symbol s in a source S. For compression purposes we require this average to be as small as possible. A theoretical value is needed, one that sets a lower bound: the closer that the average is to this bound, then the more efficient C is when used to compress data.

A theoretical lower bound is not derived from any particular code, it is derived from source symbol probabilities. For each source symbol s from a source S we measure the amount of information that s possesses. The probability of s is denoted by $P(s)$ and the unit of information [14] $I(s)$ in s is defined as:

$$I(s) = log_2 \left(\frac{1}{P(s)} \right) \text{ bits}$$

where information is measured in bits for binary encodings.

The value $I(s)$ is termed *self-information*, it is a theoretical value assigning a theoretical codeword of length $I(s)$ bits to the source symbol s. The more information s possesses, the longer $I(s)$ is, and conversely the smaller $I(s)$ is, the less information s possesses. Thus calculating the average

$$H(S) = \sum I(s)P(s) \text{ bits per source symbol}$$

for all $s \in S$ defines the theoretical lower bound, measured in bits per source symbol. This can be written as

$$H(S) = -\sum P(s)log_2P(s)$$

The value $H(S)$ is termed the Entropy [7, 14] of the source S. The entropy of S does not specify how to construct a code with average codeword length $H(S)$, it sets a theoretical lower bound. A code with average codeword length as close as possible to the entropy is termed a *compact* code.

1.4 Notation and Definitions.

Denote by N the set of natural numbers, and let $N^+ = N\backslash\{0\}$. Let $A = \{a_1, ..., a_l\}$ be a finite alphabet, where $l \in N, l > 1$. The elements in A are termed 'symbols'. Let A_n denote the set of all sequences of length n ($n \geq 1$) composed of the elements in A, and let $A^+ = \cup_{n\geq 1}A_n$. Denote the length l of a string $\bar{x}$, $\bar{x} \in A^+$, by $\mid \bar{x} \mid$, so that $\mid \bar{x} \mid = l$. The empty string λ will have length $\mid \lambda \mid = 0$. Let $A^* = A^+ \cup \{\lambda\}$. A finite subset C of A^+ is termed a code and the elements of C are termed codewords. When $A = \{0, 1\}$, C is termed a binary code. For $\bar{x}, \bar{y}, \bar{z} \in A^+$, if $\bar{x} = \bar{y}\bar{z}$, then $\bar{y}$ is a prefix of $\bar{x}$ and $\bar{z}$ is a suffix of $\bar{x}$. Merging two strings together: $\bar{x}$ and $\bar{y}$ ($\bar{x}, \bar{y} \in A^*$) is termed concatenating or appending. The appending of strings $\bar{x}$ and $\bar{y}$ is denoted by $\bar{x} \odot \bar{y}$, and a sequence of appended codewords is termed a stream. Let n_i be the number of codewords of length i in a variable length code C, then the code has length vector $(n_1, n_2, \ldots, n_M)$ where M is the maximum length of any codeword in C.

Additional notation and definitions used in this work will be presented when required.

Chapter 2

Construction of Certain Variable Length Codes.

To construct a variable length code, algorithms are applied. Three algorithms are presented here, each one creating a variable length binary code possessing certain properties.

2.1 The Huffman Code.

2.1.1 The Huffman Algorithm.

The Huffman algorithm [8] is applied to a sequence of source probabilities $p_1, p_2, ..., p_n$, with $\sum_{i=1}^{n} p_i = 1$, to generate a variable length code $\{c_1, c_2, ..., c_n\}$. Similarly, the algorithm can be applied to a sequence of source frequencies $f_1, ..., f_n$. To generate a Huffman code from the sequence of probabilities $p_1, p_2, ..., p_n$, list all probabilities in vertical ascending (or descending) order:

Probabilities

p_1

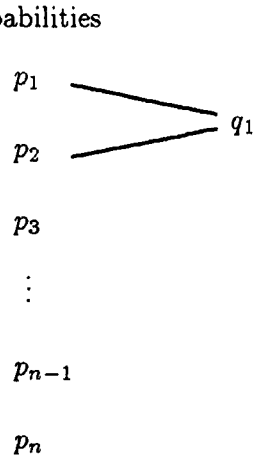
p_2

$\vdots$

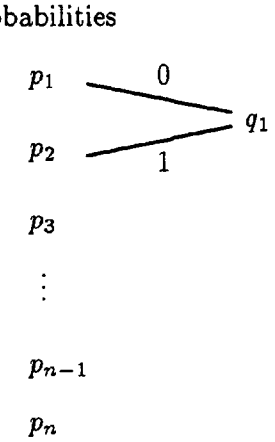
p_{n-1}

p_n

In the column "Probabilities", find the two smallest probabilities, p_1 and p_2 for instance, and calculate the sum $p_1 + p_2 = q_1$. Next, draw a line (an arc) from p_1 and p_2 to q_1:



Note that if two equal probabilities are considered, the choice can be made arbitrarily. If $p_2 \leq p_1$, label the arcs p_1 to q_1, and p_2 to q_1, by 0 and 1, respectively. Otherwise, if $p_2 > p_1$ label the lines p_2 to q_1, and p_1 to q_1, by 0 and 1, respectively:



Following the last step, a new sequence of probabilities is generated: p_1 and p_2 are omitted and replaced by the new sequence $q_1, p_3, p_4, \ldots, p_n$. The Huffman algorithm repeats: locate the two smallest probabilities x and y in $q_1, p_3, p_4, \ldots, p_n$, then draw and label arcs to the sum $x + y$, until the sum of the last two probabilities totals to 1.

The last step in the Huffman algorithm will produce a tree structure, similar to that in Figure 2.1, with each arc labelled by 0 or 1. Following the tree construction, the Huffman code is obtained. From each source symbol probability in the tree there is a sequence of edges forming a path to the value 1.0. Conversely from 1.0 there is a unique path to all n source probabilities: the path is specified as a sequence of bits, it is this sequence that forms a codeword. For example, 110 is a codeword associated with the probability p_1.

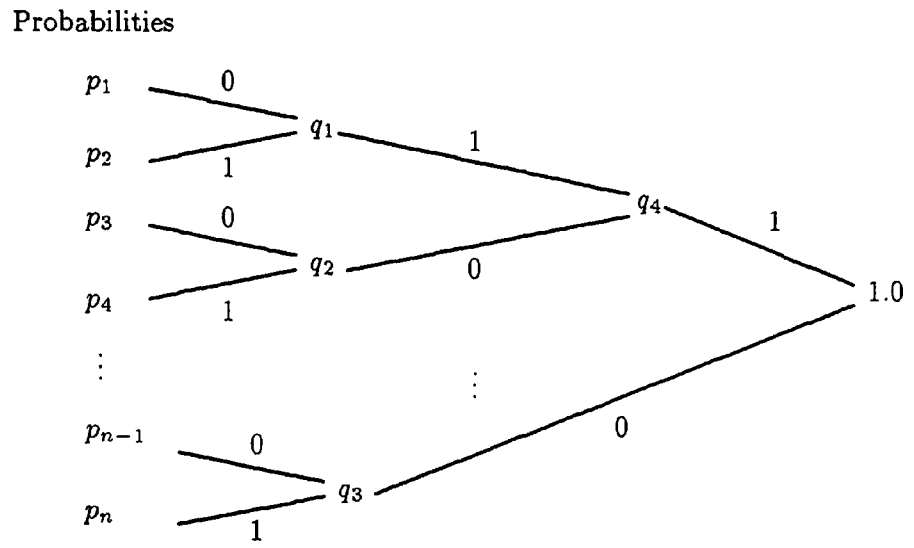


Figure 2.1: General Huffman tree.

All binary Huffman codes can be represented graphically as a binary tree. From this property all Huffman codes possess the prefix property: for any codeword c in a Huffman code C, c will not form the prefix of another codeword in C.

2.1.2 Example of Constructing a Huffman Code.

The arbitrary source symbols s_1, s_2, s_3, s_4, s_5, are to be encoded by a Huffman code. The probability of each source symbol is listed in Table 2.1:

Source Symbol	Probability
s_1	0.4
s_2	0.2
s_3	0.2
s_4	0.1
s_5	0.1

Table 2.1: Source symbol probabilities.

Apply the Huffman algorithm to the sequence of source symbol probabilities 0.1,0.1,0.2,0.2,0.4:

1. List all source symbols and respective probabilities in decreasing order:

s_1 0.4

s_2 0.2

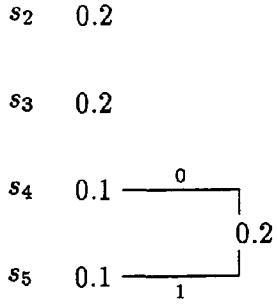
s_3 0.2

s_4 0.1

s_5 0.1

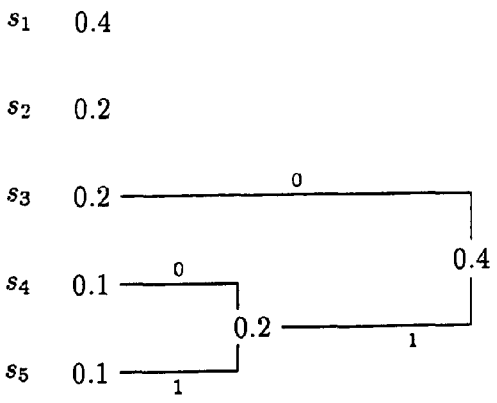
and find the two smallest probabilities.

2. The two smallest probabilities are 0.1 and 0.1. Two arcs are drawn from 0.1 and 0.1 to their sum 0.2, and each arc is labelled by a 0 or 1:



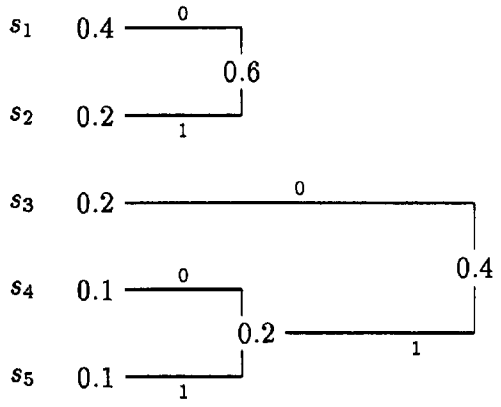
The new sequence of probabilities is: 0.2,0.2,0.2,0.4.

3. The two smallest probabilities in the new sequence are 0.2 and 0.2. Two arcs are drawn from 0.2 and 0.2 to their sum 0.4 and labelled:



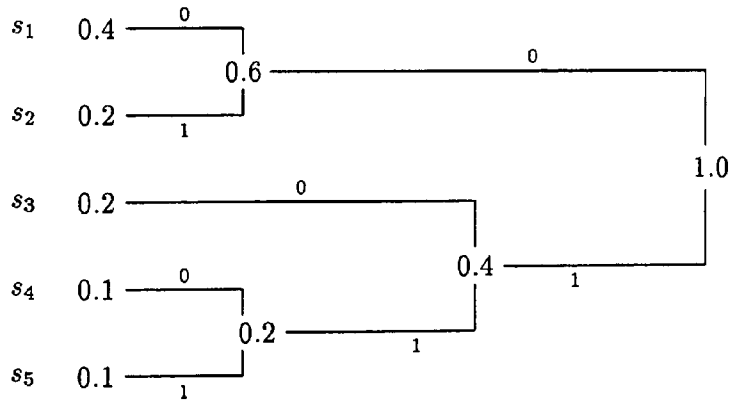
The new sequence of probabilities is: 0.4,0.2,0.4.

4. The two smallest probabilities in the new sequence are 0.4 (s_1) and 0.2. Two arcs are drawn from 0.4 and 0.2 to their sum 0.6, and labelled appropriately:



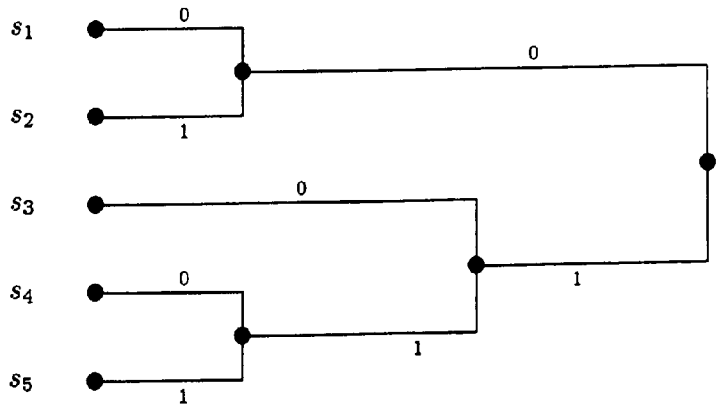
The new sequence of probabilities is: 0.6,0.4.

5. Using the remaining two probabilities 0.6 and 0.4, arcs are drawn and labelled to their sum 1.0:



The tree construction terminates when the last sum of probabilities is 1.

6. Replace all probabilities in the tree by a node '•'. For instance, beginning at the root node 1.0, replace it by '•':



This tree will be used to obtain a codeword for each source symbol.

7. Starting from the root node there are 5 unique paths, each one reaching a source symbol. With a bit (0 or 1) labelled on each arc, the 5 paths can be represented in terms of bit sequences. From the root node, traverse a path to a source symbol s_i: the bit sequence of that path is the codeword of s_i.

   The set C of paths is: $\{00, 01, 10, 110, 111\}$. Hence $C = \{00, 01, 10, 110, 111\}$ is a Huffman code for the source symbol probabilities 0.4,0.2,0.2,0.1,0.1, and each source symbol is assigned a unique codeword in Table 2.2:

Source Symbol	Probability	C
s_1	0.4	00
s_2	0.2	01
s_3	0.2	10
s_4	0.1	110
s_5	0.1	111

Table 2.2: Source symbol codewords.

The Huffman algorithm has generated a variable length code C that assigns short codewords to source symbols that are expected to be transmitted frequently, and longer codewords to source symbols that are expected to be transmitted infrequently.

2.1.3 Redundancy in Huffman Codes.

All Huffman codes are optimal, this means that the average codeword length of all Huffman codes is minimum. The Kraft inequality [17] for binary codes C

$$\sum_{i=1}^{n} 2^{-|c_i|} \leq 1$$

with $c_i \in C$, measures the amount of redundancy in a code. If the left hand side of the Kraft inequality is less than 1, then the code has redundancy, and consequently the average codeword length of C is not minimum. Otherwise, when the left hand side of Kraft's inequality equals 1, the code possesses minimum redundancy and the average codeword length of C is minimum.

Huffman coding is a good method of encoding data compactly: the average codeword length is always minimum and therefore close to the entropy of a source. For example, the Huffman code for the source S in Table 2.2 above has an average codeword length of 2.2 that is close to the entropy 2.123.

The Huffman code is not the only optimal code, other optimal variable length codes are available. Variable length codes that are optimal and not necessarily derived from the Huffman algorithm are termed Huffman equivalent in this work.

2.1.4 Huffman Codes in Practice.

Huffman coding is used to construct codewords for any number of source symbols given that the symbol probabilities (or frequencies) are known. For example, in Table 2.3, a Huffman code is presented which encodes letters in the English alphabet, including the symbol '*space*'.

Encoding source symbols using a Huffman code does not require the use of space to separate each codeword for identification: an encoded sequence of symbols will be one continuous string of zeros and ones – a bit stream. For example, the sentence "a day at the zoo" encoded with C from Table 2.3 produces the bit stream:

000011010110000000111111000000110110011011111001100001111111100100010

Decoding a bit stream will produce the original message: reading from left to right, bit by bit, replace the first recognizable codeword with its source symbol equivalent, until the end of the stream. To decode the bit stream representing "a day at the zoo" decode the first recognizable codeword:

a11010110000000111111000000110110011011111001100001111111100100010

then the second codeword, until the end of the sequence to produce the original message:

a day at the zoo

The decoding rule is an algorithm denoted by D. With the prefix property inherent in all Huffman codes, D is a suitable decoder. For instance, in a bit stream composed of codewords from a Huffman code, no codeword is the prefix of another, hence D will correctly decode, without the need of space to separate and identify each codeword.

Source Symbol	Source Symbol Probability	Huffman Code C
'*space*'	0.1859	110
e	0.1031	100
t	0.0796	0110
a	0.0642	0000
o	0.0632	0010
i	0.0575	0100
n	0.0574	0111
s	0.0514	1010
r	0.0484	1110
h	0.0467	1111
l	0.0321	00110
d	0.0317	10110
u	0.0228	00010
c	0.0218	01010
f	0.0208	010110
m	0.0198	000110
w	0.0175	001110
y	0.0164	001111
p	0.0152	010111
g	0.0152	101110
b	0.0127	101111
v	0.0083	0001110
k	0.0049	00011110
x	0.0013	000111110
j	0.0008	0001111110
q	0.0008	00011111110
z	0.0005	00011111111

Table 2.3: Huffman code for the English alphabet, with '*space*'.

For any sequence of source symbols encoded by a Huffman code, the decoder D will always produce the original sequence. However if the encoded sequence was transmitted over a noisy channel and subject to channel errors, causing corruption, the decoder cannot identify and amend the corrupted parts.

2.1.5 Synchronization.

Implementations of Huffman codes cannot ignore the presence of channel errors and their effects. The effect of channel errors on Huffman codes can vary, depending upon the specific code used. Channel errors can cause a minimal or significant amount of corruption.

When an error affects a bit stream $\bar{s}$ the result is a new stream $\bar{s}'$. Let $\bar{s}$ denote the sequence of bits:

$$| b_1b_2b_3 | b_4b_5b_6b_7 | b_8b_9 | b_{10}b_{11}b_{12} |$$

using '|' to identify each codeword $b_1b_2b_3$, $b_4b_5b_6b_7$, b_8b_9, and $b_{10}b_{11}b_{12}$. If $\bar{s}$ was affected by error, causing the inversion of bits b_5 and b_8 to produce b'_5 and b'_8, then one possibility of $\bar{s}'$ is the sequence of codewords

$$| b_1b_2b_3 | b_4b'_5 | b_6b_7b'_8b_9 | b_{10}b_{11}b_{12} |$$

Comparing $\bar{s}$ and $\bar{s}'$

$$\bar{s} = | b_1b_2b_3 | b_4b_5b_6b_7 | b_8b_9 | b_{10}b_{11}b_{12} |$$

$$\bar{s}' = | b_1b_2b_3 | b_4b'_5 | b_6b_7b'_8b_9 | b_{10}b_{11}b_{12} |$$

consider how the decoder D will decode the new stream $\bar{s}'$. D will decode correctly the first codeword $b_1b_2b_3$ in $\bar{s}'$, because $b_1b_2b_3$ is also the first codeword in $\bar{s}$. In this case D is synchronized with $\bar{s}'$. However as D decodes the codeword $b_4b'_5$, synchronization is lost because the true sequence is suppose to be b_4b_5, the prefix of $b_4b_5b_6b_7$. Synchronization is regained after decoding $b_6b_7b'_8b_9$.

To illustrate the effects of synchronization loss on a coded piece of text, let C denote the Huffman code assigned to the alphabet $\{A, B, C, D, E, F, G, H, I\}$ in Table 2.4. Encode and transmit the source sequence $ACFHF$, that is: 10000001001100010. Prior to decoding the bit stream 10000001001100010, apply a bit inversion error to the second bit: 1$\underline{1}$000001001100010 – as underlined. The received bit stream 1$\underline{1}$000001001100010 decodes to the sequence $DCBHF$. Comparing the original data to the received data:

Alphabet	Huffman Code C	Alphabet	Huffman Code C
A	10	F	0010
B	010	G	0011
C	000	H	0110
D	110	I	0111
E	111		

Table 2.4: Huffman code for an alphabet.

Original: ACFHF

Received: DCBHF

the tail end HF of the received data is part of the original data sent, whereas the prefix part DCB of $DCBHF$ is not the original sequence. During the receipt of 11000001001100010, as the decoder D began decoding the first codeword in the bit stream 11000001001100010 synchronization was lost and remained so until the receipt of the codeword 010 representing symbol 'B': correct decoding commenced prior to receipt of the codeword representing 'H'.

Hence despite the presence of one error, part of the original data was salvaged. This was possible due to synchronous properties inherent in Huffman codes. When part of a bit stream is corrupted the Huffman code can possess properties limiting the affect of errors, thus allowing parts of the original data to be preserved.

2.1.6 Synchronizing Sequences.

A synchronizing sequence is a series of bits in a bit stream that forces synchronization. The specific sequence $\bar{b}$ in a stream will force a decoder D that is unsynchronized to regain synchronization after $\bar{b}$ is decoded. For example, the code $C = \{00, 01, 10, 110, 111\}$ has a synchronous sequence 0110; which can be formed by $01 \odot 10$ or $00 \odot 110$. Assigning source symbols to each codeword in C, as specified in Table 2.5 and encoding the letter sequence *eebcda* produces the sequence:

111111011011000

Inverting the second bit gives the sequence: 101111011011000. This can now be decoded and

Source Symbol	C
a	00
b	01
c	10
d	110
e	111

Table 2.5: Assigning a Huffman code.

compared to the original source sequence sent:

Original: *eebcda*
Received: *cecdda*

The string *cecdda* illustrates that synchronization was lost at the start of decoding the first codeword in 101111011011000 until receipt of the synchronous string 0110 represented by *bc*, thus resynchronizing the decoder with the original bit stream prior to receipt of the suffix 11000 (after decoding 0110) and therefore preserving the original tail end *da*.

2.1.7 Synchronizing Codewords.

Ferguson & Rabinowitz [4] have proved that some binary Huffman codes contain a codeword that resynchronizes the decoder, regardless of the synchronization slippage preceding that codeword. A code achieving this is termed *synchronous*. A synchronous codeword operates like a synchronous sequence except that the sequence is a codeword.

A synchronous variable length code C is formally defined in [4] as:

if there is a codeword $c = c_1c_2...c_n$ *in* C *satisfying the following two conditions:*

- *for all* $x = x_1x_2...x_m$ *in* C *such that* $m > n$ *and* c *is a substring of* x, *we have* $c_1c_2...c_n = x_{m-n+1}...x_m$ *but* $c_1c_2...c_n \neq x_ix_{i+1}...x_{i+n-1}$ *for any* $i \neq m - n + 1$;
- *for any* $j < n$ *such that* $c_1c_2...c_j$ *can be written as a suffix, the sequence* $c_{j+1}c_{j+2}...c_n$ *is a string of codewords;*

then C *is synchronous, and* c *is termed a synchronizing codeword of* C.

It is worth noting that more than one synchronizing codeword can exist in a synchronous code. For example $C_{example}$ ={00,10,11,010,011} is synchronous with two synchronizing codewords 010 and 011. We will identify synchronous codewords by marking each one by an asterix "*", so that $C_{example} = \{00, 10, 11, 010^*, 011^*\}$.

2.1.8 Strong Equivalence of Huffman Codes.

Let C be a Huffman code and $\bar{x} = x_1x_2...x_n$ denote any binary n–tuple, that is a sequence of bits 0 and 1. Define a new Huffman code C$\bar{x}$ by a process of twisting the tree representing C at node $x_1x_2...x_n$. The operation C$\bar{x}$ is defined in [4] as:

for all $c_1...c_j \in C$, *if* $j > n$ *and* $c_1...c_n = x_1...x_n$,
then $c_1...c_nc'_{n+1}c'_{n+2}...c'_j$ *is in* $C\bar{x}$
(with c'_i *denoting complementation), otherwise* $c_1...c_j \in C\bar{x}$.

The procedure $C\bar{x}$ can be iterated, such that $C\bar{x}_1,\bar{x}_2,...,\bar{x}_k$= $(...((C\bar{x}_1)\bar{x}_2)...)\bar{x}_k$. Hence two codes C_1 and C_2 are said to be strongly equivalent if there is a sequence $\bar{x}_1,...,\bar{x}_k$ so that C_2=$(...((C\bar{x}_1)\bar{x}_2)...)\bar{x}_k$. For example, C_1={1,01,001,0001,0000} and C_2={1,00,011,0101,0100} are strongly equivalent because C_2=$(C_1\bar{x}_1)\bar{x}_2$, for $\bar{x}_1 = 0$ and $\bar{x}_2 = 01$, as illustrated by Figure 2.2.

If C is obtainable by the Huffman algorithm, then a code that is strongly equivalent to C is also obtainable by the same algorithm; with different choices for zero and one at some points in the procedure.

The set of codes related by the strong equivalence property is termed a strong equivalence class, and each code in this class has the same length vector. This suggests that if a synchronous code exists in the equivalence class, a nonsynchronous code belonging to the same class can be replaced by the synchronous code – if a synchronous code is required. However, some classes do not contain synchronous codes. An example of a Huffman code that is not synchronous and does not possess a synchronous encoding in its equivalence class is given in [4] and tabulated in Table 2.6.

When a synchronous code is needed and a non-synchronous Huffman code does not possess a synchronous code in its equivalence class, then a Huffman equivalent code may be sought, one that is not derived from the Huffman algorithm and is strongly equivalent to a synchronous code. The use of such Huffman equivalent codes avoids the need to search for strongly equivalent synchronous codes.

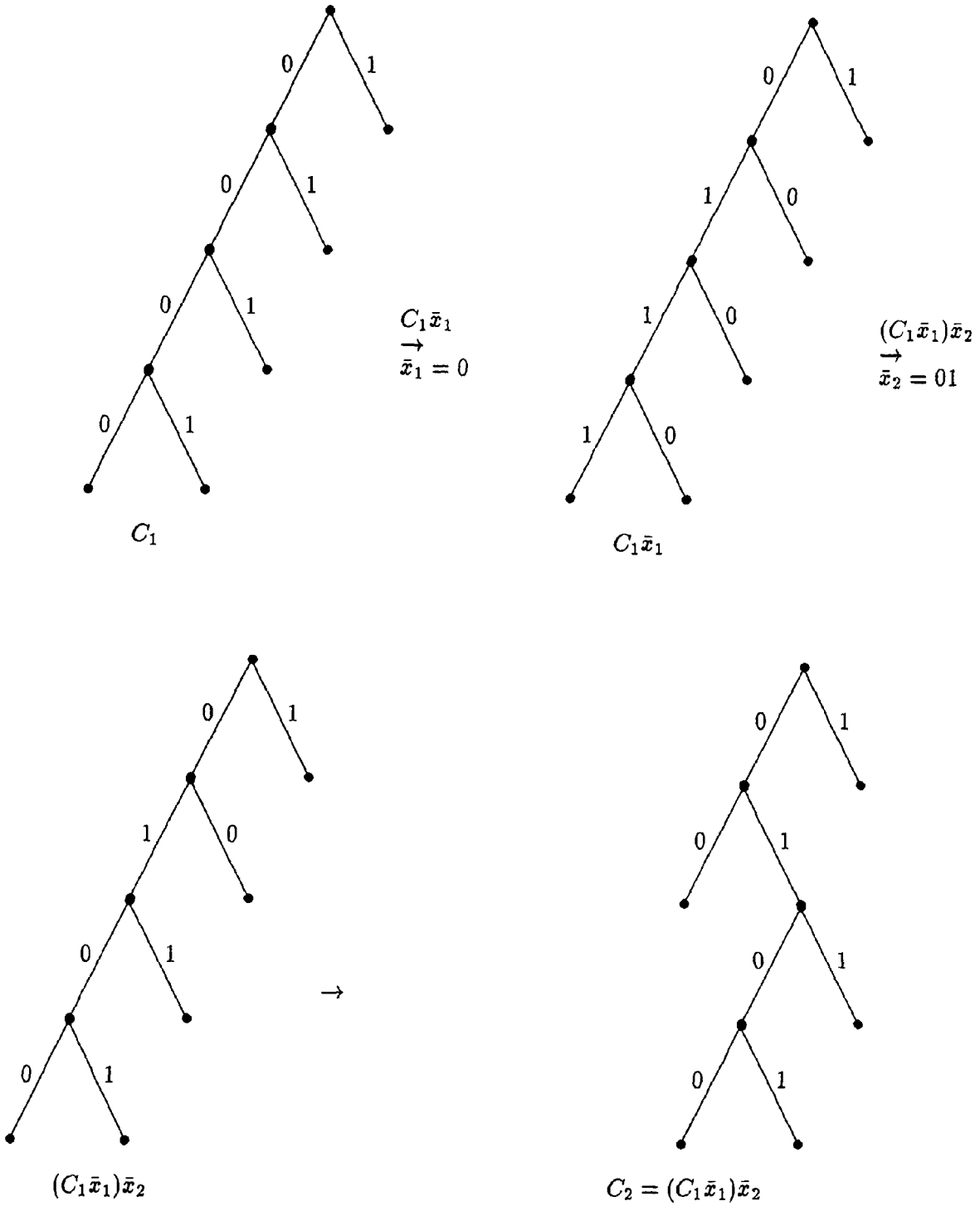


Figure 2.2: Strong equivalence example.

Probability	C_3
0.3	10
0.3	11
0.1	000
0.1	001
0.1	010
0.1	011

Table 2.6: A non-synchronous Huffman code.

2.1.9 Weak Equivalence of Huffman Codes.

When a non–synchronous code is not strongly equivalent to a synchronous one, an alternative approach is to search for a different synchronous code as a substitute. In this circumstance the codes are *weakly equivalent*. Two variable length codes are weakly equivalent when the sets of codeword lengths and their multiplicities are the same for both codes and the codes are not strongly equivalent (i.e. they have the same length vector). Weak equivalence suggests that when a code C is not synchronous and does not possess strong equivalence to a synchronous code, C may be substituted by another code C_s, provided that the codeword lengths in C_s are equal to those in C.

For example, $C_2 = \{01, 11, 000, 001, 100, 101^*\}$ is synchronous but not strongly equivalent to $C_3 = \{10, 11, 000, 001, 010, 011\}$, therefore C_2 and C_3 are weakly equivalent. The distribution of lengths in both C_2 and C_3 match, hence C_3 can be substituted by C_2 if a synchronous code is needed.

2.2 A Huffman Equivalent Code Designed to Force Resynchronization.

2.2.1 The Algorithm.

The algorithm given in [3] constructs a synchronous code C that is Huffman equivalent, with the following properties:

- the shortest codeword has length $m > 1$;
- the shortest synchronizing codeword is of length $r = m + 1$ and is denoted by $c_1 \ldots c_r$;
- the shortest synchronizing codewords are specified to be 01_{r-1} or $01_{r-2}0$; or alternatively as 10_{r-1} or $10_{r-2}1$.

The algorithm involves constructing a binary tree from a given length vector $(n_1, n_2, ..., n_M)$ satisfying the Kraft inequality, with equality. One such vector can be obtained from a known Huffman code, for instance. The tree will have specific node types, and it is these that dictate how a tree is constructed.

There are four node types, each of which occurs during the algorithm. To classify each type, define the string $x_1x_2...x_n$ to be an arbitrary path in an arbitrary binary tree, starting from the root node to an end node:

- c–node: a node is a c–node (or a codeword node) if its path $x_1x_2 \ldots x_n$ is either $c_2 \ldots c_r$ or $x_{n-r+1} \ldots x_n = c_1c_2 \ldots c_r$.
- d–node: a node is a d–node if its path $x_1x_2 \ldots x_n$ has $x_{n-k+1} \ldots x_n = c_1 \ldots c_k$ for some k, $2 \leq k < r$.
- 1–node: a node is a 1–node if it is neither a c–node nor a d–node, and its path ends in a 1.
- 0–node: a node is a 0–node if it is neither a c–node nor a d–node, and its path ends in a 0.

For example, if 0110 is taken to be the short synchronizing codeword then 0110, 110 and $b_1 \ldots b_i0110$ are c–nodes, 01 and 011 are examples of d–nodes.

In pseudo-code, the algorithm in [3] is stated:

Select the synchronizing codeword $c_1c_2...c_r$.
At each length i *in turn repeat:*
Terminate all length i c*–nodes.*
Extend all length i d*–nodes.*
If $i \geq m$ *then terminate any* 0*–nodes or* 1*–nodes as required.*
Extend the remaining 0*–nodes and* 1*–nodes*
Until no non-terminated nodes remain.

The advantage of applying this algorithm to construct a variable length code is that synchronizing codewords are generated. If for a given length vector a code exists that contains a synchronizing codeword of length one longer than the shortest codeword in the code, then the algorithm will always generate it (or an equivalent one). The Huffman algorithm does not guarantee such a code.

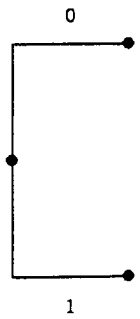
The algorithm in [3] does not produce a code for all length vectors satisfying Kraft's inequality, with equality. When the algorithm fails for a given length vector $(n_1, n_2, ..., n_M)$, a substitute length vector with average codeword length close to that of the required length vector can be used.

In this thesis codes generated by the algorithm given in this section will always be termed Huffman equivalent.

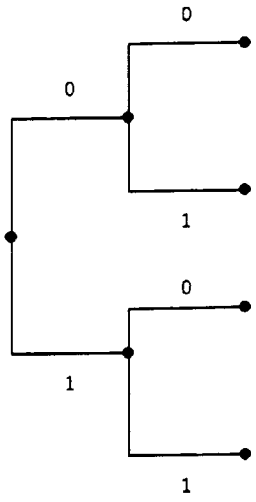
2.2.2 Code Construction Example.

Construct a Huffman equivalent code with length vector (0,1,4,4) and synchronous codeword 010*, by terminating 0–nodes whenever possible:

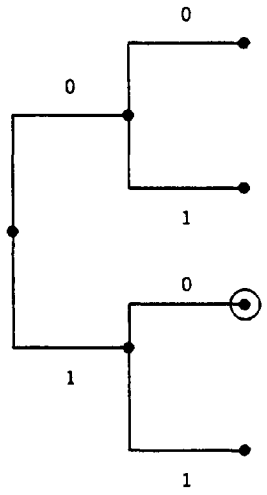
1. The initial state of the algorithm is a simple binary tree:



2. Extend all end nodes in the simple binary tree:

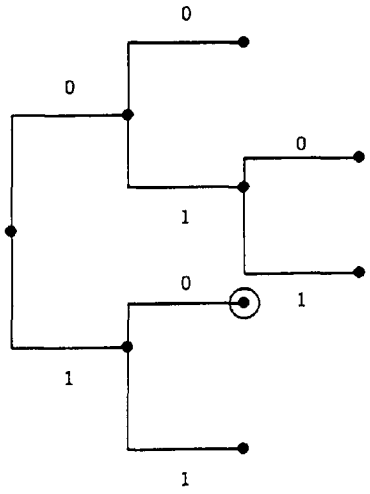


Search for c–nodes and terminate each one: one c–node '10', the suffix of '010', is found. The end node of the path '10' is thereby terminated – as circled:



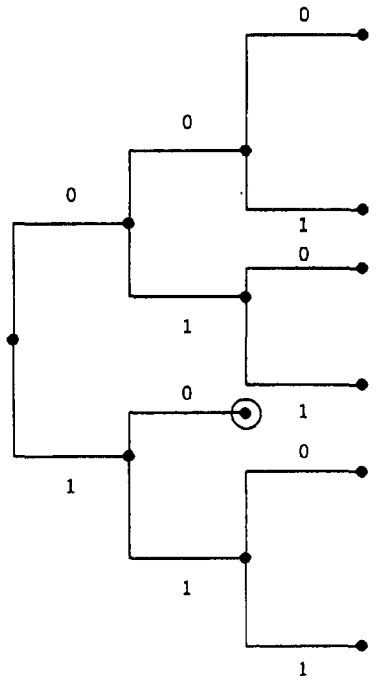
One codeword '10' of length 2 is obtained.

3. Extend all d–nodes. There is one d–node present: the node at the end of path '01' originating from the root node:

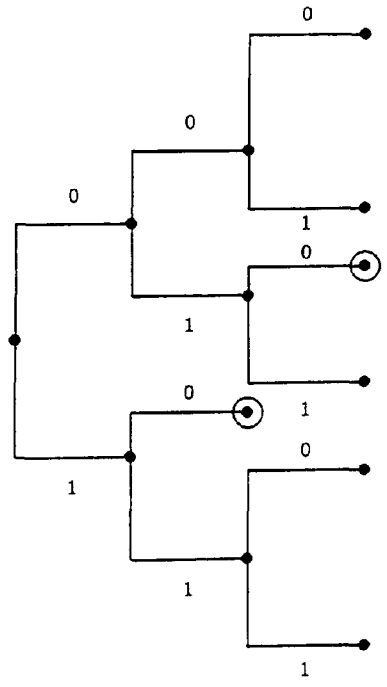


4. The length vector (0,1,4,4) specifies that one codeword of length 2 is required, this is the codeword '10'.

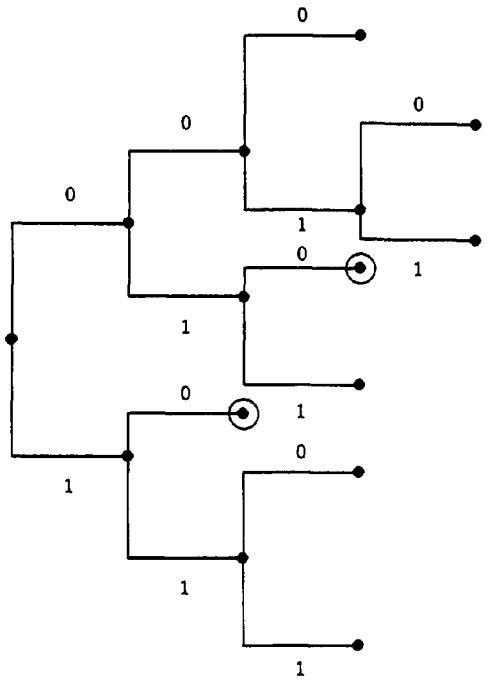
5. Extend the remaining nodes in our binary tree:



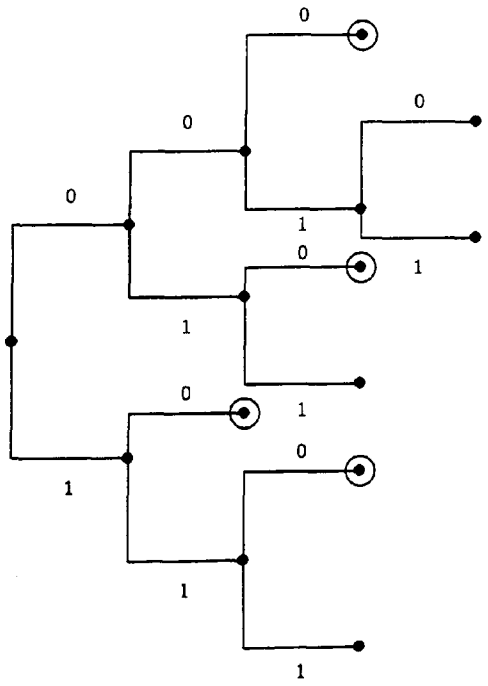
6. Terminate all c–nodes. Terminate the path '010':



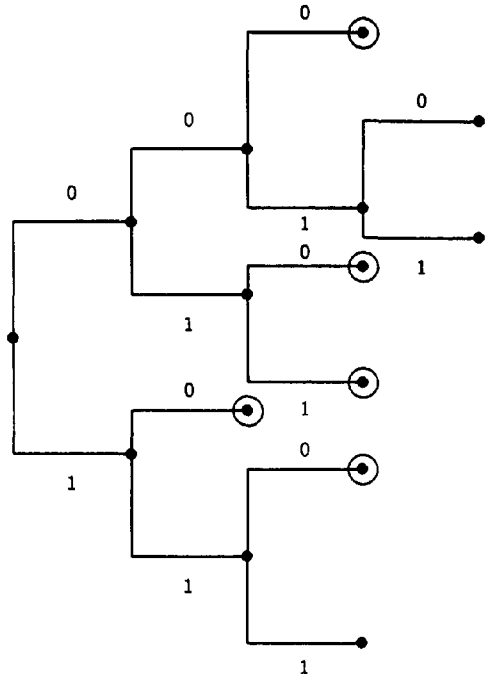
7. Extend all d–nodes. The d–nodes are paths ending in '01':



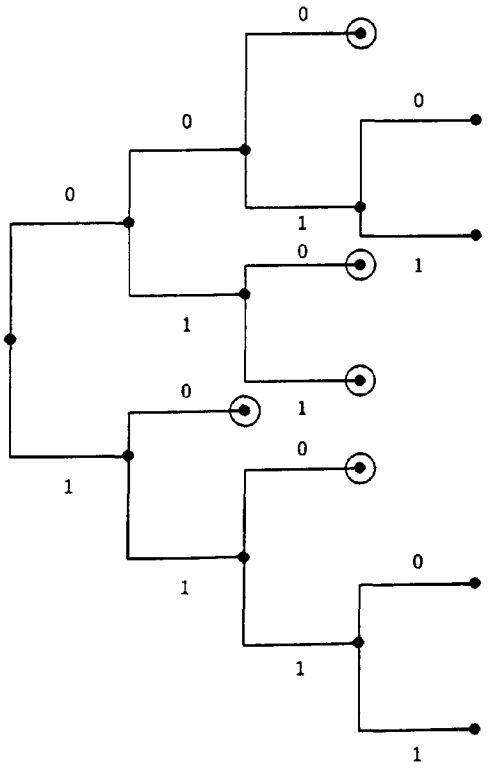
8. The length vector (0,1,4,4) specifies that four codewords of length 3 are required. One codeword '010' has been found, two more codewords are obtained by terminating the necessary 0–nodes:



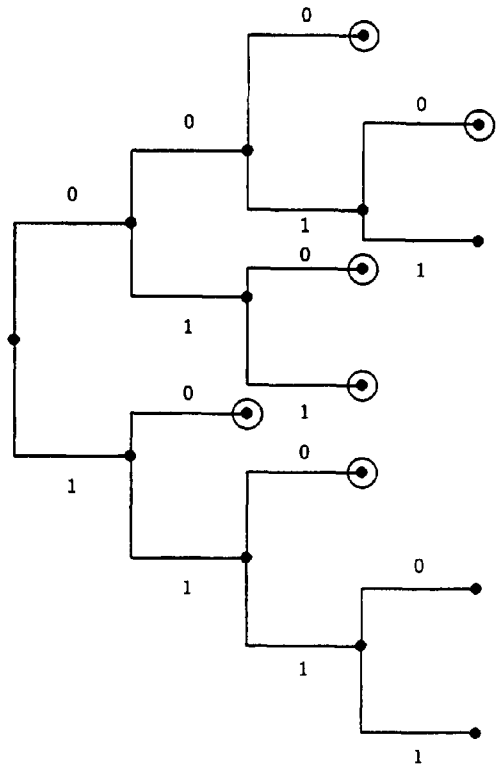
One more codeword is needed, this means terminating a 1–node. Two 1–nodes remains, one of these is terminated:



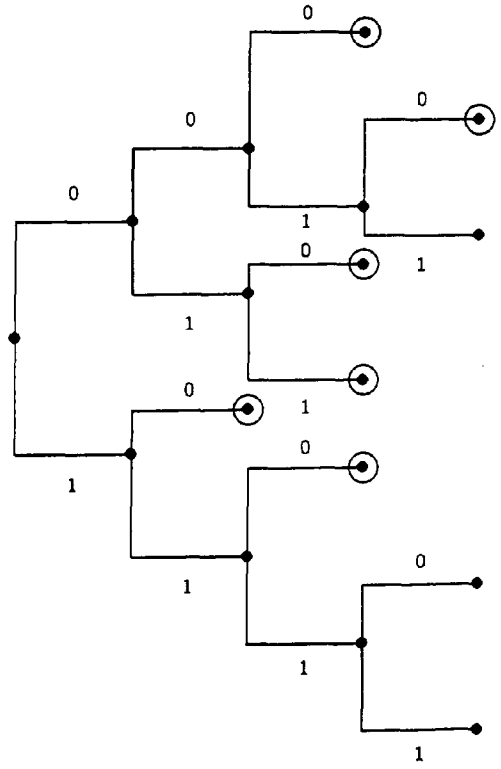
Extend the non–terminated nodes that remains:



9. Terminate all c–nodes. Terminate the path '0010', with the suffix '010':

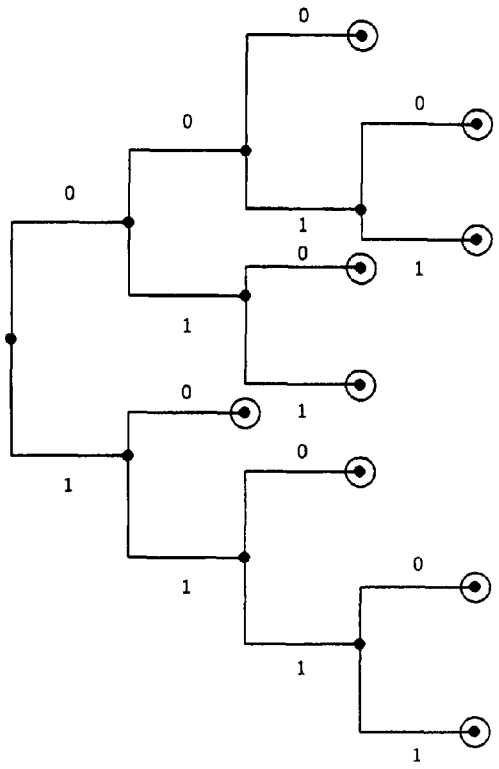


10. At this stage of the algorithm there are no d–nodes to extend. There is one terminated c–node, two 1–nodes and one 0–node:



11. The length vector (0,1,4,4) specifies that four codewords of length 4 are required. One codeword '0010' has already been obtained, to obtain three other codewords it is safe to terminate the

nodes that remain:



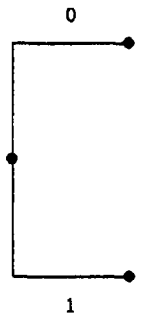
All nodes in the tree are terminated, therefore the algorithm terminates.

The binary tree is used to obtain the codewords: each codeword is a path from the root node to an end node; the path specified by a binary sequence is a codeword. The code in this example is $\{000, 0010^*, 0011, 010^*, 011, 10, 110, 1110, 1111\}$.

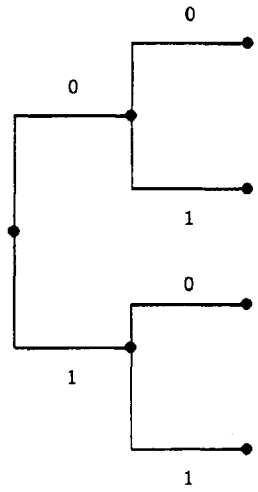
Using the length vector (0,0,2,8,5,4,3,1,1,2) that satisfies Kraft's equality, a larger Huffman equivalent code can be constructed, that contains a short synchronizing codeword 0110^*, by terminating 0–nodes when possible. The algorithm will generate the code $\{000, 110, 0110^*, 0100, 0010, 1010, 1000, 1110, 0111, 1111, 00110^*, 10110^*, 10010, 01010, 00111, 100110^*, 010110^*, 101110, 100111, 0101110, 1011110, 0101111, 10111110, 101111110, 1011111110, 1011111111\}$ containing 5 synchronizing codewords.

Applying the algorithm to the same length vector (0,0,2,8,5,4,3,1,1,2), with short synchronizing codeword 0110^*, does not generate a synchronous code if 1–nodes are terminated when possible. The algorithm also fails to construct a synchronous code with a short synchronizing codeword 0111^* when terminating 0–nodes or when terminating 1–nodes. For example, apply the algorithm to the length vector (0,0,2,8,5,4,3,1,1,2), using a short synchronizing codeword 0111^* and terminating 0–nodes:

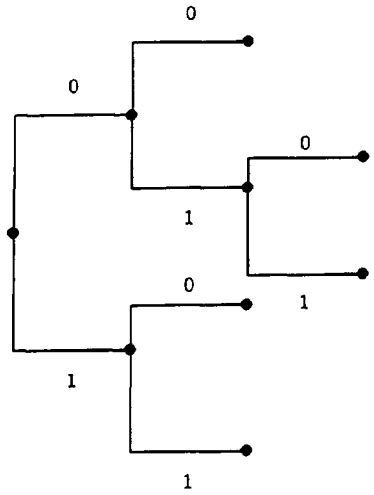
1. Begin the algorithm with a simple binary tree:



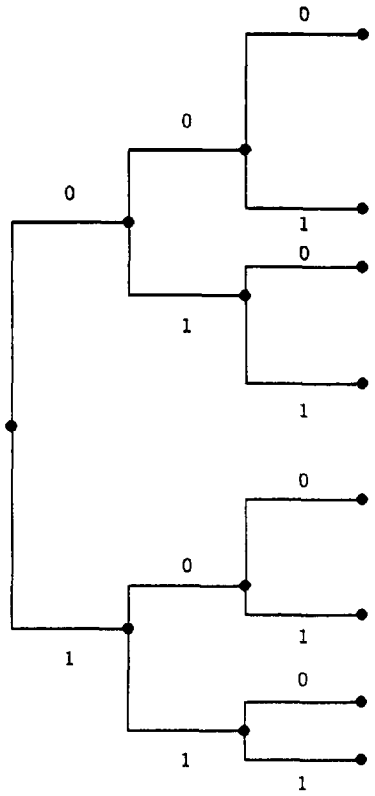
2. Extend all end nodes in the simple binary tree:



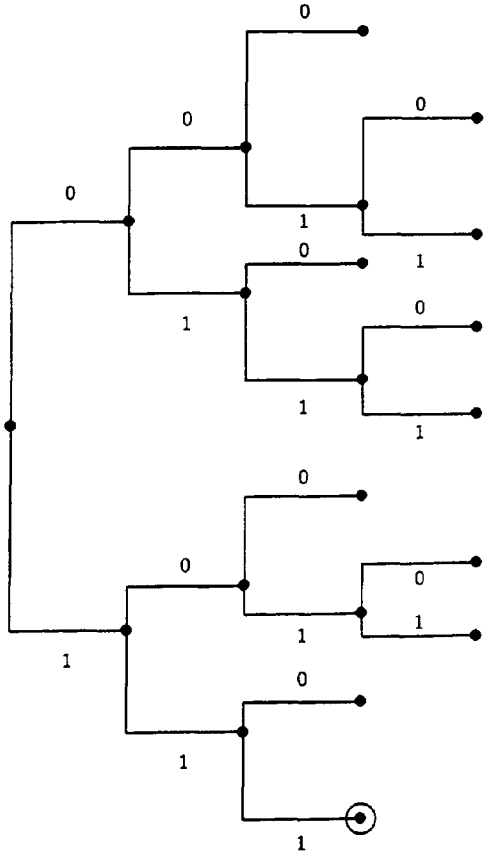
3. Extend all d–nodes. There is one d–node present: the node at the end of path '01' originating from the root node:



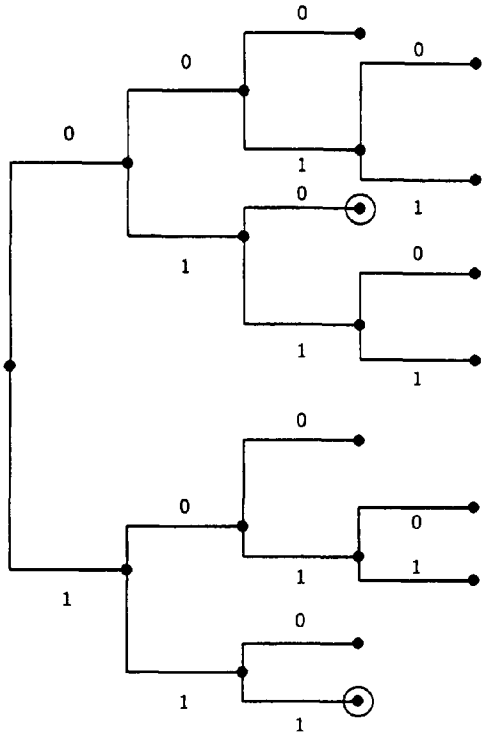
4. Extend the nodes that remain:



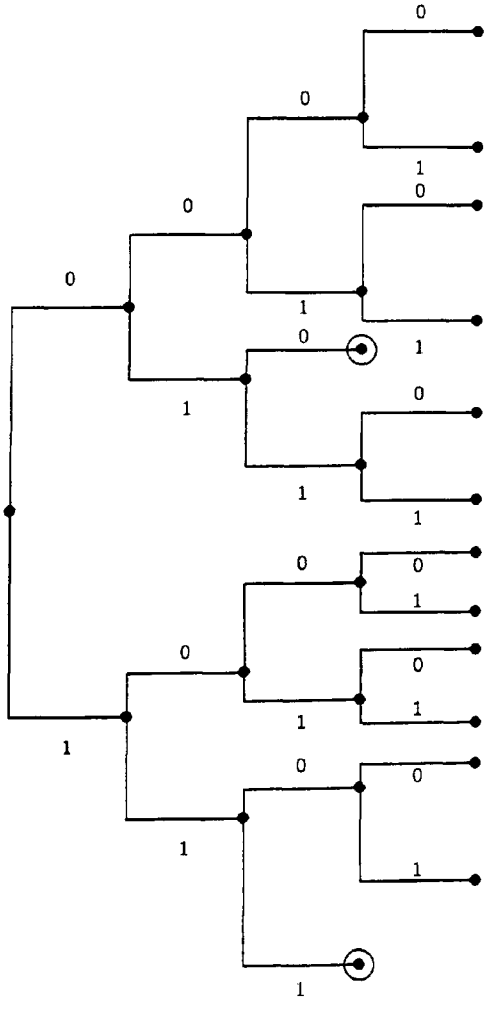
then terminate the c–node 111 and extend all d–nodes:



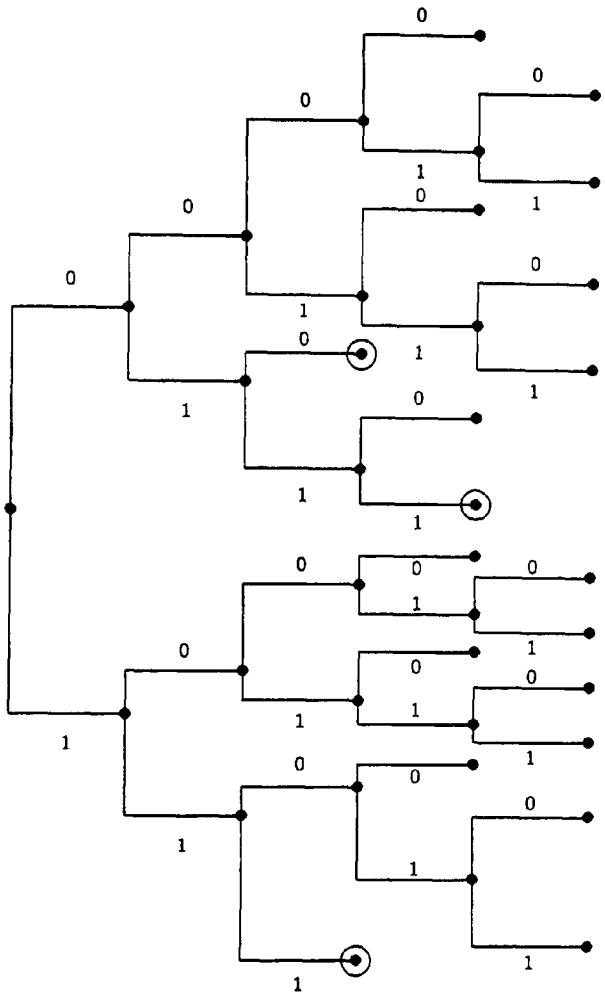
Two codewords of length 3 are required, terminate a 0–node to obtain the second codeword:



5. Extend the nodes that remain:



then terminate all c–nodes and extend all d–nodes:



At this stage of the algorithm, eight nodes with paths of length 4 are required, however only seven nodes can be terminated: the algorithm fails in this case.

In [3] a theorem is given stating that if there exists a synchronous code C with length vector $(n_1, \ldots, n_M)$ and synchronizing codeword of length $r = m + 1$, where $m > 1$, then a synchronous code can be generated by the algorithm. The code constructed may be C or an equivalent one. To construct the code, the algorithm is applied by chosing $01_{r-2}0$ as the synchronizing codeword and extending 1–nodes (or equivalently terminating 0–nodes) where possible.

In [12] a more general version of the algorithm can be found.

2.2.3 Algorithm Properties.

Let C denote a synchronous code obtained from the Huffman equivalent algorithm. The length vector $(n_1, \ldots, n_M)$ of C is specified: $n_i = 0$ for $i < m$ and $n_m \geq 1$ for some $m > 1$. The shortest synchronizing codeword has length $r = m + 1$.

It does not matter which particular 0–nodes or 1–nodes are terminated when applying the algorithm, although the choice of how many 0–nodes or 1–nodes to terminate at each stage of the algorithm is important. For example, the Huffman equivalent codes C_1 and C_2 on page 35, Table 2.7, are constructed using the synchronizing codeword $01_{r-2}0$ and length vector (0,0,2,8,4,7,1,1,1,1,2). It is observed that C_2 has one more synchronizing codeword than C_1, this is because during the construction of code C_2 a 0–node of path length 5 from the root node was extended rather than terminated.

The type of synchronizing codeword used in the algorithm can also determine the number of synchronizing codewords that are generated: using the synchronizing codeword 01_{r-1} in the algorithm can sometimes generate a code with more synchronizing codewords as opposed to a code constructed from the codeword $01_{r-2}0$. The Huffman equivalent codes C_3 and C_4 below are constructed from the length vector (0,1,4,4) and synchronizing codewords $01_{r-2}0$ and 01_{r-1} respectively, by terminating 0–nodes when possible; C_4 contains one more synchronizing codeword than code C_3:

C_3	C_4
10	11
010*	011*
011	010
000	000
110	100
0010*	0011*
0011	1011*
1110	1010
1111	0010

C_1	C_2
110	110
100	100
0110*	0110*
0000	0000
0010	0010
0100	0100
0111	0111
1010	1010
1110	1110
1111	1111
00110*	00110*
10110*	10110*
00010	00111
01010	01010
010110*	010110*
000110*	000110*
001110	000100
001111	000111
010111	010111
101110	101110
101111	101111
0001110	0001010
00011110	00010110*
000111110	000101111
0001111110	0001011100
00011111110	00010111010
00011111111	00010111011

Table 2.7: Comparison of two synchronous codes.

When the shortest codeword has length $m \geq 3$, the code constructed by algorithm [3] will contain 2^{j-1} length $m+j$ synchronizing codewords for all j such that $2 \leq j \leq m-1$. Note that each of these forced codewords has as its suffix the chosen synchronizing codeword. The codes C_1 and C_2 illustrate this; each containing additional synchronizing codewords of length 5. Also note that longer synchronizing codewords exist although these are not guaranteed by the algorithm.

2.2.4 Presence of Synchronizing Sequences.

The algorithm that constructs a synchronous code also permits synchronizing sequences. The presence of codewords 1_{r-1} or $1_{r-2}0$ in a bit stream can form a synchronous sequence, depending on its location. The formation of a synchronizing sequence would depend upon the preceding bit: if the bit '0' was a prefix to both codewords 1_{r-1} and $1_{r-2}0$, giving the sequences '...01_{r-1}...' or '...$01_{r-2}0$...' in a bit stream, then a synchronizing sequence is formed because 01_{r-1} or $01_{r-2}0$ are synchronizing codewords.

For example, let $\bar{s}$ denote the sequence of codewords:

$$x_1x_2x_3\underline{1_{r-1}}x_4x_5x_6x_7\underline{1_{r-1}}x_8x_9\underline{1_{r-1}}x_{10}$$

containing the shortest codeword 1_{r-1} (as underlined) that is the suffix of the short synchronizing codeword 01_{r-1}. Transmit $\bar{s}$ and assume that each codeword 1_{r-1} in $\bar{s}$ is received error free. Following transmission, if the suffix part of x_3 ends with a '0', so that $x_3 = y_30$ for instance, then the sequence $\bar{s}$ is:

$$x_1x_2y_3\underline{01_{r-1}}x_4x_5x_6x_7\underline{1_{r-1}}x_8x_9\underline{1_{r-1}}x_{10}$$

and the decoder D automatically synchronizes when the string 01_{r-1} is received, because 01_{r-1} is a synchronizing codeword. Similarly if $x_7 = y_70$ and $x_9 = y_90$, D automatically synchronizes on receipt of 01_{r-1} in the sequence $\bar{s}$:

$$x_1x_2y_3\underline{01_{r-1}}x_4x_5x_6y_7\underline{01_{r-1}}x_8y_9\underline{01_{r-1}}x_{10}$$

Thus any code constructed by the algorithm given in [3] contains synchronizing codewords and synchronizing sequences. These may be optimised by careful selection of the synchronizing codeword and of the termination or extension of 0 and 1-nodes.

2.3 T–Codes.

The T–Code algorithm [16] generates a variable length prefix code that is uniquely decodable. The code is termed a T–Code and there are two types: simple and generalised.

2.3.1 The Simple T–Code Algorithm: Simple T–augmentation.

The simple T–code algorithm is initially applied to a known code C to construct a larger code. C must be a prefix code, thus a known Huffman code can be used to construct T–codes, or we can use $C = \{0, 1\}$ for example. The process of deriving one code from another is termed an *augmentation*. For example, an augmentation that constructs the T–code $C_{(c_j)}$ from C with $c_j \in C$ is depicted:

C	$C_{(c_j)}$
c_1	c_1
c_2	c_2
$\vdots$	$\vdots$
c_{j-1}	c_{j-1}
c_j	c_{j+1}
c_{j+1}	$\vdots$
$\vdots$	c_n
c_n	c_jc_1
	$\vdots$
	c_jc_j
	$\vdots$
	c_jc_n

To derive $C_{(c_j)}$ from C, list all codewords in C, except for c_j, then list again all codewords in C, having first appended them to c_j. Formally this is defined

$$C_{(c_j)} = C\backslash\{c_j\} \cup c_jC$$

with $C\backslash\{c_j\}$ denoting the removal of c_j from C and c_jC denoting the appending operation of all codewords in C to c_j.

Augmentation can be applied successively and indefinitely. For example selecting c_{j+1} from $C_{(c_j)}$ the T-code $C_{(c_j,c_{j+1})}$ is constructed:

C	$C_{(c_j)}$	$C_{(c_j,c_{j+1})}$
c_1	c_1	c_1
c_2	c_2	c_2
$\vdots$	$\vdots$	$\vdots$
c_{j-2}	c_{j-2}	c_{j-1}
c_{j-1}	c_{j-1}	c_{j+2}
c_j	c_{j+1}	$\vdots$
c_{j+1}	$\vdots$	c_n
$\vdots$	c_n	c_jc_1
c_n	c_jc_1	$\vdots$
	$\vdots$	c_jc_{j-1}
	c_jc_{j-1}	c_jc_j
	c_jc_j	c_jc_{j+1}
	c_jc_{j+1}	$\vdots$
	$\vdots$	c_jc_n
	c_jc_n	$c_{j+1}c_1$
		$c_{j+1}c_2$
		$\vdots$
		$c_{j+1}c_{j-2}$
		$c_{j+1}c_{j-1}$
		$c_{j+1}c_{j+1}$
		$c_{j+1}c_{j+2}$
		$\vdots$
		$c_{j+1}c_n$
		$c_{j+1}c_jc_1$
		$\vdots$
		$c_{j+1}c_jc_{j-1}$
		$c_{j+1}c_jc_j$
		$c_{j+1}c_jc_{j+1}$
		$\vdots$
		$c_{j+1}c_jc_n$

Here $C_{(c_j,c_{j+1})}$ has been constructed from C by two augmentations; selecting c_j and c_{j+1} to be the prefix words in each augmentation, respectively. There is no rule governing which codewords are to be used for an augmentation, it is a matter of preference.

In general, given C, the set resulting from n levels of T–augmentation with prefixes $c_i, c_j, \ldots, c_n$, is denoted by $C_{(c_i,c_j,\ldots,c_n)}$ or alternatively by $C_{(c_i,c_j,\ldots,c_{n-1})(c_n)}$ where $C_{(c_i,c_j,\ldots,c_{n-1})(c_n)} = C_{(c_i,c_j,\ldots,c_n)}$.

2.3.2 Example of a Simple T–augmentation Process.

As an example of a T–augmentation process, let $C = \{0, 1\}$. The following table will show that $C_{(0,1,101)} = \{00, 01, 11, 100, 10100, 10101, 10111, 101100, 101101\}$:

C	$C_{(0)}$	$C_{(0,1)}$	$C_{(0,1,101)}$
0	1	00	00
1	00	01	01
	01	11	11
		100	100
		101	10100
			10101
			10111
			101100
			101101

2.3.3 Node Reduction and Extension.

To construct a T–code with a specific number of codewords, augmentation alone cannot guarantee to produce the required number of codewords. For example let $C = \{0, 1\}$, if 5 codewords are required then $C_{(1,10)}$ is one possibility:

C	$C_{(1)}$	$C_{(1,10)}$
0	0	0
1	10	11
	11	100
		1010
		1011

If 4 codewords were needed then any augmentation starting from $C = \{0, 1\}$ cannot generate this number. One option is to begin with a different starting set C that contains a different number of codewords. However no code C exists in this case. The guaranteed method of producing a T–code with the specified number of codewords is *node reduction*.

Node reduction involves finding any two codewords c_x, c_y of chosen length m from a code $C_{(c_i,c_j,\ldots,c_n)}$. c_x and c_y must have the same prefix $\bar{z}$ of length $m-1$ and a different suffix of length 1:

$$c_x = \bar{z}b$$

$$c_y = \bar{z}b'$$

denoting the inverted bit of b by b'. After locating c_x and c_y, both are removed from $C_{(c_i,c_j,\ldots,c_n)}$ and replaced by $\bar{z}$, thus reducing the number of codewords in $C_{(c_i,c_j,\ldots,c_n)}$ by 1.

For example, applying node reduction to $C_{(1,10)}$

$C_{(1,10)}$
0
11
100
1010
1011

by removing the two codewords 1011 and 1010 from $C_{(1,10)}$ and adding the prefix 101 to $C_{(1,10)}$, produces a new code with 4 codewords:

$C_{(1,10)}$
0
11
100
101

Node reduction is a process that can remove a codeword from a code, the code remains a prefix code. The process can be applied successively to obtain a code with the required number of codewords. With successive augmentations and node reductions, a simple T–code of any size can be constructed.

Alternatively, *node extension* increases the number of codewords in a T–code by 1. To apply a node

extension operation to a codeword c in a T–code C, form the codewords $c0$ and $c1$ by appending 0 and 1 to the suffix of c. A new T–code is then formed by removing c from C and including codewords $c0$ and $c1$.

2.3.4 Generalised T–augmentation.

The simple T–code algorithm can be extended to produce *generalised* T–codes [15]. Denoting c_j^m to be m concatenations of a codeword c_j:

$$c_j^m = c_j c_j \ldots c_j$$

Formally the generalised T–code is defined as:

$$C_{(p)}^{(k)} = \{\cup_{i=0}^{k}\ p^i(C \backslash p)\} \cup \{p^{k+1}\}$$

A generalised augmentation applied to construct $C_{(p)}^{(k)}$ is illustrated in Table 2.8. This code was constructing from C to form a generalised T–code $C_{(c_j)}^{(k)}$ where $c_j \in C$.

As an example, select the codeword 1 from $C = \{1, 01, 00\}$ to construct the generalised T–code $C_{(1)}^{(3)}$ in one augmentation:

C	$C_{(1)}^{(3)}$
1	01
01	00
00	101
	100
	1101
	1100
	1111
	11101
	11100

Observing that $C_{(c_j)}^{(k)}$ equals the simple T–code $C_{(c_j)}$ when $k = 1$, then $C_{(c_i,c_j,\ldots,c_n)}^{(1,1,\ldots,1)} = C_{(c_i,c_j,\ldots,c_n)}$ which is a particular case of the general form $C_{(c_i,c_j,\ldots,c_n)}^{(u,v,\ldots,w)}$.

C	$C^{(k)}_{(c_j)}$
c_1	c_1
$\vdots$	$\vdots$
c_{j-1}	c_{j-1}
c_j	c_{j+1}
c_{j+1}	$\vdots$
$\vdots$	c_n
c_n	c_jc_1
	$\vdots$
	c_jc_{j-1}
	c_jc_{j+1}
	$\vdots$
	c_jc_n
	$c_j^2c_1$
	$\vdots$
	$c_j^2c_{j-1}$
	$c_j^2c_{j+1}$
	$\vdots$
	$c_j^2c_n$
	$\vdots$
	$c_j^kc_1$
	$\vdots$
	$c_j^kc_{j-1}$
	c_j^{k+1}
	$c_j^kc_{j+1}$
	$\vdots$
	$c_j^kc_n$

Table 2.8: Generalised T-code, one augmentation.

2.3.5 T–Code Construction Example.

As an example of constructing a T–code from simple and generalised T–augmentations, let $C = \{0, 1\}$ be the starting set used to construct the T–code $C^{(1,3,1)}_{(0,00,1)}$:

C	$C^{(1)}_{(0)}$	$C^{(1,3)}_{(0,00)}$	$C^{(1,3,1)}_{(0,00,1)}$
0	1	1	01
1	00	01	001
	01	001	0001
		0001	00001
		00001	000001
		000001	0000001
		0000001	00000000
		00000000	00000001
		00000001	11
			101
			1001
			10001
			100001
			1000001
			10000001
			100000000
			100000001

The T–code $C^{(1,3,1)}_{(0,00,1)}$ consists of 17 codewords, to obtain 16 codewords apply node reduction operations. Thus codewords 100000001 and 100000000 are removed from $C^{(1,3,1)}_{(0,00,1)}$ and replaced by codeword 10000000, forming the T–code:

01,	11,	001,	101
0001,	1001,	00001,	10001
100001,	0000001,	1000001,	000001
00000000,	10000001,	00000001,	10000000

2.3.6 Synchronization of T–codes.

The format of the simple and generalised T–code is one that encourages synchronization to occur. For instance, the construction of $C_{(c_j)}$ from C forms a variable length code:

$$C_{(c_j)} = \{c_1, c_2, ..., c_{j-1}, c_{j+1}, ...c_n, c_jc_1, c_jc_2, ..., c_jc_j, ..., c_jc_n\};$$

$C_{(c_j)}$ can be written as a union of sets:

$$S_1 = \{c_1, c_2, ..., c_{j-1}, c_{j+1}, ..., c_n\}$$

and

$$S_2 = \{c_jc_1, c_jc_2, ..., c_jc_j, ..., c_jc_n\}$$

The codewords contained in set S_2 are each composed of a prefix c_j and suffix c_i (for $1 \leq i \leq n$). The suffix c_i is a codeword (except when $c_i = c_j$). The set of all codewords $S_2 \backslash \{c_jc_j\}$ will encourage synchronization due to the presence of codewords as suffixes. To illustrate this, let $\bar{s}$ denote the string of codewords

$$| x_1x_2x_3 | y_1y_2y_3y_4 | c_jc_i | z_1z_2 |$$

each separated by "|", with $x_1x_2x_3, y_1y_2y_3y_4, c_jc_i, z_1z_2 \in C_{(c_j)}$.

In $\bar{s}$ a bit inversion error occurs to y_2, producing y_2':

$$x_1x_2x_3y_1y_2'y_3y_4c_jc_iz_1z_2$$

If the string $y_1y_2'y_3y_4$ is composed of one or more codewords, then synchronization is not lost, and correct decoding begins on receipt of codeword c_jc_i. Similarly, if the string $y_1y_2'y_3y_4c_j$, with arbitrary suffix c_j, is composed of one or more codewords, synchronization will be regained because the string $y_1y_2'y_3y_4c_jc_i$ will be composed of two or more codewords, and correct decoding begins prior to receipt of codeword z_1z_2.

2.3.7 Optimal T–codes.

There are two ways to construct simple or generalised T–codes: using a construction with or without node reductions or extensions. In either case it is unknown whether an optimal T–code can always be constructed. In Chapter 6, some examples of optimal T–codes are constructed through the application of node reduction and/or extension operations. However, the general case remains to be proved.

Modifying a T–code, by applying node reduction and/or extension operations, may affect the inherent synchronization properties. Extending or reducing codewords in a T–code will produce a different code, it is unknown exactly what effect such operations have: synchronization properties may improve or worsen.

Chapter 3

Markov Modelling.

Markov models will feature in later chapters, it is therefore necessary to introduce some of the basic theory [6] at this stage by means of an example.

3.1 Example of a Markov Model.

There are 6 abstract states, the collection of which is termed a *state space*: $\{E_1, E_2, ..., E_6\}$. The current state (or initial/starting state) is $x_0 = E_4$, as identified by an arrow "$\Downarrow$":

$$\begin{array}{cccccc} & & & \Downarrow & & \\ E_1 & E_2 & E_3 & E_4 & E_5 & E_6 \end{array}$$

The rules of this example are that:

1. the arrow moves one state at a time;
2. in one step, the arrow can: move left once; move right once or remain stationary;
3. if the arrow arrives at states E_1 or E_6, then the arrow remains there permanently;
4. the probabilities of the arrow changing states in one step are specified as:
   (a) 1/2=$P(\textit{moving left})$;
   (b) 1/3=$P(\textit{moving right})$;
   (c) 1/6=$P(\textit{stationary})$;

The initial state of this example can be described by a row vector $\bar{x}_0$ termed the 'initial row vector', in which the i–th component of $\bar{x}_0$ contains the probability that the arrow is initially at state E_i. For this example, the arrow is initially stationed at state E_4. Therefore $\bar{x}_0 = (0, 0, 0, 1, 0, 0)$, with probability $p(E_4) = 1$ because E_4 is the starting state.

From the initial state E_4, the arrow can move in one step to one of two states E_5, E_3 or remain stationary at state E_4. The probability of changing states, from the initial state, in one step is described by a row vector $\bar{x}_1$. The vector $\bar{x}_1$ details the conditional probabilities:

$$\bar{x}_1 = (P(E_1 \mid E_4), P(E_2 \mid E_4), P(E_3 \mid E_4), P(E_4 \mid E_4), P(E_5 \mid E_4), P(E_6 \mid E_4))$$

$$= (0, 0, 1/2, 1/6, 1/3, 0)$$

Interpreting $\bar{x}_1$, the i–th entry in $\bar{x}_1$ is the probability of moving from the initial state E_4 to state E_i in one step. The probability of moving from one state to another is termed a 'transition probability'. For example, the transition probability $P(E_4 \rightarrow E_5)$ in one step is $1/3$, and the transition probability $P(E_4 \rightarrow E_6)$ in one step is 0, which means that it is not possible for the arrow to move from state E_4 to state E_6 in one step.

Similar to $\bar{x}_1$, the vector $\bar{x}_2$ details the two step transition probabilities $P(E_4 \rightarrow E_i)$ for all $E_i \in \{E_1, E_2, E_3, E_4, E_5, E_6\}$. For instance, the transition probability $P(E_4 \rightarrow E_6) = 1/9$ in two steps and the transition probability $P(E_4 \rightarrow E_1) = 0$ that is not possible in two steps. Interpretation of $\bar{x}_2$ reveals that the arrow is more likely to remain stationed at state E_4 after two steps, this is because $P(E_4 \rightarrow E_4)$=13/36 is the largest probability in $\bar{x}_2$.

To calculate the vector $\bar{x}_2$, $\bar{x}_2$ can be derived from the vector $\bar{x}_1$: for each non–zero probability in $\bar{x}_1$, second step transition probabilities are calculated and combined to form $\bar{x}_2$. To form $\bar{x}_2$ from $\bar{x}_1$, assign to each non–zero probability in $\bar{x}_1$ a vector $\bar{x}_{(1,i)}$, where i denotes an ith probabilty. Thus with three non–zero probabilities in $\bar{x}_1$, three vectors $\bar{x}_{(1,1)}, \bar{x}_{(1,2)}, \bar{x}_{(1,3)}$ are formed:

$$\bar{x}_{(1,1)} = (0, 0, 0, 0, 1/3, 0)$$

$$\bar{x}_{(1,2)} = (0, 0, 0, 1/6, 0, 0)$$

$$\bar{x}_{(1,3)} = (0, 0, 1/2, 0, 0, 0).$$

For each vector $\bar{x}_{(1,1)}, \bar{x}_{(1,2)}, \bar{x}_{(1,3)}$, calculate the second step transition probabilities, that are $\bar{x}_{(2,1)}$, $\bar{x}_{(2,2)}$, and $\bar{x}_{(2,3)}$ respectively. For example, the vector $\bar{x}_{(1,1)}$ is used to construct $\bar{x}_{(2,1)}$:

$$\bar{x}_{(1,1)} = (0,0,0,0,1/3,0)$$

$$\bar{x}_{(2,1)} = (0,0,0,(1/3)(1/2),(1/3)(1/6),(1/3)(1/3))$$

$$= (0,0,0,1/6,1/18,1/9).$$

The ith entry in $\bar{x}_{(2,1)}$ is a specific probability. For instance, given that the initial state is E_4, the probability of moving from state E_4 to state E_5, then from state E_5 to state E_6 is: $1 \times 1/3 \times 1/3$.

Similar to the construction of $\bar{x}_{(2,1)}$, vectors $\bar{x}_{(2,2)}$ and $\bar{x}_{(2,3)}$ are constructed from vectors $\bar{x}_{(1,2)}$ and $\bar{x}_{(1,3)}$, respectively:

$$\bar{x}_{(1,2)} = (0,0,0,1/6,0,0)$$

$$\bar{x}_{(2,2)} = (0,0,(1/6)(1/2),(1/6)(1/6),(1/6)(1/3),0)$$

$$= (0,0,1/12,1/36,1/18,0)$$

$$\bar{x}_{(1,3)} = (0,0,1/2,0,0,0)$$

$$\bar{x}_{(2,3)} = (0,(1/2)(1/2),(1/2)(1/6),(1/2)(1/3),0,0)$$

$$= (0,1/4,1/12,1/6,0,0)$$

It follows that the second step transition probabilities from an initial state E_4 is interpreted as: $\bar{x}_{(2,1)}$ or $\bar{x}_{(2,2)}$ or $\bar{x}_{(2,3)}$, that is (using vector addition):

$$\bar{x}_2 = \bar{x}_{(2,1)} + \bar{x}_{(2,2)} + \bar{x}_{(2,3)} = (0,1/4,1/6,13/36,1/9,1/9).$$

Calculating the vector $\bar{x}_k$ after k steps becomes increasingly difficult if k is large. Furthermore, in this example the initial state is E_4: other states could have been chosen instead. If E_3 was chosen to be the initial state, then vectors $\bar{x}_0 = (0,0,1,0,0,0)$ and $\bar{x}_1 = (0,1/2,1/6,1/3,0,0)$ are the initial and one step vectors, respectively. By considering all initial states, and the transition probabilities after k steps, a Markov model is formed.

To form a Markov model for this example all one–step state transition probabilities are represented in matrix form T. In T, let each row denote a one step vector:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/6 & 1/3 & 0 & 0 & 0 \\ 0 & 1/2 & 1/6 & 1/3 & 0 & 0 \\ 0 & 0 & 1/2 & 1/6 & 1/3 & 0 \\ 0 & 0 & 0 & 1/2 & 1/6 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The element $t_{(i,j)}$ in row i column j of T is a one–step transition probability: $t_{(i,j)}$ is the probability of moving from the initial state E_i to state E_j in one step. For example, if the initial state was E_4, then $t_{(4,5)}$=1/3 is the one–step transition probability from state E_4 to state E_5.

The matrix T is termed a transition matrix. The transition matrix is used to obtain the n–step transition probabilities using the power T^n: the entry $t_{(i,j)}$ in T^n states the probability of moving from state E_i to state E_j after n–steps. Hence, probabilities such as $P(E_i \rightarrow E_j)$ after n–steps are provided by T^n.

For example, consider the matrix T^2:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 7/12 & 7/36 & 1/9 & 1/9 & 0 & 0 \\ 1/4 & 1/6 & 13/36 & 1/9 & 1/9 & 0 \\ 0 & 1/4 & 1/6 & 13/36 & 1/9 & 1/9 \\ 0 & 0 & 1/4 & 1/6 & 7/36 & 7/18 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The entry $t_{(i,j)}$ in T^2 is the two step transition probability from the initial state E_i to state E_j. In particular, the value $t_{(4,5)}$ in T^2 is the two step transition probability from initial state E_4 to state E_5.

In T^2, observe that row 4 is equal to the two–step vector $\bar{x}_2$. To obtain the vector $\bar{x}_2$ from T^2, apply the product $\bar{x}_0 T^2$, with $\bar{x}_0 = (0,0,0,1,0,0)$:

$$\bar{x}_0 T^2 = (0, 1/4, 1/6, 13/36, 1/9, 1/9) = \bar{x}_2$$

and similarly for $\bar{x}_0 = (0,0,0,0,1,0)$ the product $\bar{x}_0 T^2$ is equal to the two–step vector from initial

state E_5:

$$\bar{x}_0 T^2 = (0, 0, 1/4, 1/6, 7/36, 7/18)$$

Hence the matrix T^n can be used to state the probability of the arrow being in one of six different states after n–steps.

3.2 The General Markov Model.

A Markov model is described by a transition matrix T^n, it is a model comprised of states and probabilities. The matrix T^n conveniently provides n–step state transition probabilities, that are used to study processes in the long term. The states used in Markov theory are arbitrary, they are defined only when a specific model is constructed. However, in a specific model it is important to recognize how the transition matrix T is originally constructed.

For example, consider two matrices T_1 and T_2, the one–step transition probabilities defined in T_1 can differ from those in another matrix T_2 that is also used to model the same processes as T_1. The states defined for T_1 will be the same states used by T_2, yet T_1^n may prove to be a better model than T_2^n, or vice versa, in terms of long term predictability.

Employing the matrix T^n, the vector $\bar{x}_0 T^n$ can be obtained. The vector $\bar{x}_0 T^n$ details the n–step transition probabilities from an initial state i to all states in the state space; with the ith entry in $\bar{x}_0$ equal to 1, and other entries equal to 0.

We will use Markov modelling in subsequent chapters to estimate error recovery.

Chapter 4

Two Methods of Calculating the Error Recovery of Variable Length Codes.

4.1 Defining the Error Recovery.

4.1.1 Decoder States.

A decoder D that decodes a received bit stream $\bar{s}$ can switch between states. The decoder has no mechanism of knowing its own state. For example, suppose a bit stream $\bar{s}$ is transmitted and $\bar{s}$ is received error free, then the decoder is uniformly synchronized with $\bar{s}$; from the beginning of decoding the first codeword in $\bar{s}$ to the decoding of the last codeword in $\bar{s}$. However, the decoder does not *know* if it is synchronized or not.

If a transmission error occurs in $\bar{s}$, then D will change states. For example, $\bar{s}$ is a bit stream composed of codewords c_1, c_2, c_3, c_4, c_5, let $\bar{s} = c_1 c_2 c_3' c_4 c_5$ with c_3' denoting the codeword c_3 in error. Following the decoding of string $c_1 c_2$, D begins decoding the string c_3'. As D reads the first bit of c_3', the state of the decoder is denoted by I: the initial state prior to a transmission error.

If c_3' is a string of one or more codewords, then D will not lose synchronization: D will change its state to the synchronization recovery state S after reading the last bit of c_3' and prior to reading the first bit of c_4. It is interpreted that it took one word c_3' to resynchronize, even if c_3' is composed of two or more codewords. For a variable length code C, the probability of this occurring is denoted

by $P_1(I \rightarrow S)$: the one–step transition probability of D from state I to synchronization recovery state S.

Alternatively the string c_3' can equal fx. The prefix part f of c_3' either equals a string of one or more codewords or the null string λ, and the suffix part x of c_3' is some prefix word of a codeword. If $c_3' = fx$ then $\bar{s}$ equals:

$$c_1 c_2 f x c_4 c_5.$$

As D reads the first bit of string fx, the initial state of D is I, and after reading the last bit of fx (prior to reading the first bit of c_4), the state of D is denoted by x. The probability of this occurring is denoted by $P_1(I \rightarrow x)$: the one–step transition probability from state I to error state x.

Denote by $\{f_i\}$ the set of error states that are the prefix words of codewords in C. Hence for the variable length code C, one–step transition probabilities from state I to error state f_i are denoted by $P_1(I \rightarrow f_i)$.

With $\bar{s} = c_1 c_2 f x c_4 c_5$ and the present state of D denoted by x, the next state of D will depend upon codeword c_4. When D reads the last bit of c_4 and xc_4 is a string of one or more codewords, then the state of D changes to the synchronization recovery state S with one–step probability $P_1(x \rightarrow S)$. For a variable length code C, $P_1(f_i \rightarrow S)$ denotes the one–step transition probability of D from error state f_i to synchronization recovery state S.

When xc_4 is not a string of one or more codewords, then xc_4 will equal some string: with a prefix part composed of one or more codewords (or λ) and a suffix part equal to a prefix word x'. With the current state of D equal to the error state x, the decoder will change its state to error state x' after reading the last bit of xc_4 and before reading the first bit of c_5, with one–step probability $P_1(x \rightarrow x')$.

For a variable length code C, $P(f_i \rightarrow f_j)$ denotes the transition probability of D from error state f_i to error state f_j.

4.1.2 n–Step Decoder–State Transition Probabilities.

When an error affects a codeword, we are interested in the probability of decoder D remaining synchronized or regaining synchronization when it is lost. For example, let $\bar{s} = c_1 c_2 c_3 c_4 c_5$ be a bit stream composed of variable length codewords c_1, c_2, c_3, c_4, c_5. Suppose the string $\bar{s}$ is subject to an error, the error occurs to c_3, producing the string c_3' in $\bar{s}$:

$$\bar{s} = c_1 c_2 c_3' c_4 c_5.$$

Consider the decoder states of D as $\bar{s}$ is decoded. The state of the decoder D is I as D reads the first bit of c_3'. As D reads the last bit of c_3', the state of D changes to Y – a synchronized or error state Y. The probability $P_1(I \rightarrow Y)$ that D changes state from I to Y is termed a *one–step* state transition probability because the probability $P(I \rightarrow Y)$ is derived from *one* codeword in error.

Assume that the one–step state transition probability $P_1(I \rightarrow Y)$ is from initial state I to error state Y, and let D decode the next word c_4 in $\bar{s}$. As D reads the last bit of c_4 the state of D is Y' – a synchronized or error state Y'. The probability $P_2(I \rightarrow Y')$ that D changes state from I to Y' is termed a *two–step* state transition probability, because two codewords were read by D: one codeword in error and one error free codeword.

We can define n–step state transition probabilities. Let C denote a variable length code with $c \in C$. If a channel error affected the codeword c in an arbitrary bit stream $\bar{b}$ composed of codewords from C, the decoder D has an initial state I as it reads the first bit of c. D continues to read the subsequent $n-1$ codewords in $\bar{b}$, that may or may not be affected by error. The probability that D synchronizes within n codewords is then denoted by $P_n(I \rightarrow S)$.

There are many ways of constructing n–step state transition probabilities. The methods used in this work will derive n–step state transition probabilities from one–step state transition probabilities by means of Markov modelling.

4.1.3 The State Transition Diagram.

When one–step state transition probabilities are known, a graph G drawn to represent the one–step state transition probabilities is termed an *error state transition diagram*, where all nodes in G correspond to the states defined by a variable length prefix code C; with a set of prefix words F. Hence each node in G is an error state $f \in F$, an initial state I or a synchronized state S, all linked by arcs labelled by a one–step state transition probability. The state transition diagram G will have the general form shown in Figure 4.1.

In G there are arcs *from* node I to all state nodes $f \in F$ and S; there are no arcs entering node I. Node I is termed a *source*. There is also one arc from each error state node and state node I, *into* node S. There are no arcs from S. Node S is termed a *sink* and corresponds to the synchronization recovery state. Hence the state transition diagram G represents graphically all

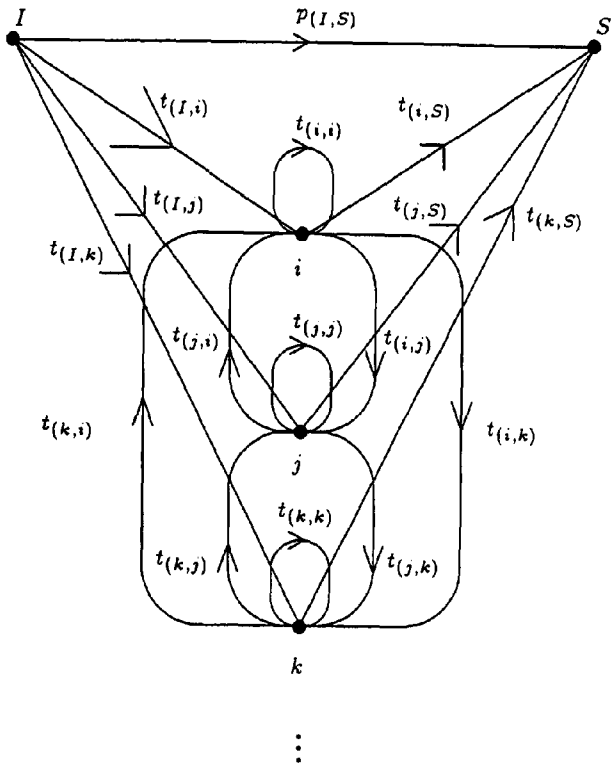


Figure 4.1: The general state transition diagram.

one–step state transition probabilities. The diagram will be used to calculate n–step state transition probabilities $P_n(I \to S)$.

4.1.4 The State Transition Matrix.

The transition matrix is another means of representing one–step state transition probabilities. For a variable length code C with prefix words F, a matrix M of order $(N+1) \times (N+1)$ is used. For N prefix words and one decoder state I, M will resemble the matrix:

$$\begin{pmatrix} p_{(0,0)} & q_{(0,1)} & q_{(0,2)} & \cdots & q_{(0,N)} \\ s_{(1,0)} & t_{(1,1)} & t_{(1,2)} & \cdots & t_{(1,N)} \\ s_{(2,0)} & t_{(2,1)} & t_{(2,2)} & \cdots & t_{(2,N)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ s_{(N,0)} & t_{(N,1)} & t_{(N,2)} & \cdots & t_{(N,N)} \end{pmatrix}$$

with entries defined as follows:

1. $P(I \rightarrow S) = p_{(0,0)}$: the transition probability from the initial state I to synchronization recovery state S

2. $P(I \rightarrow p_i) = q_{(0,i)}$ for each $p_i \in F$, $0 \leq q_{(0,i)} \leq 1$, $i = 1, \ldots, N$: the transition probability from the initial state I to an error state p_i

3. $P(p_i \rightarrow S) = s_{(i,0)}$ for each $p_i \in F$, $0 \leq s_{(i,0)} \leq 1$, $i = 1, \ldots, N$: the transition probability from an error state p_i to the synchronization state S

4. $P(p_i \rightarrow p_j) = t_{(i,j)}$ for each $p_i, p_j \in F$, $i, j = 1, \ldots, N$: the transition probability among error states.

The transition matrix M will be used to construct n–step state transition probabilities $P_n(I \rightarrow S)$.

4.1.5 The Theoretical Error Recovery.

When a decoder D loses synchronization, resynchronization may occur after one or more codewords. The number of codewords required to resynchronize varies, a theoretical measure is therefore required to predict the average number of codewords required to resynchronize when D loses synchronization.

Let X denote a discrete random variable. X is the number of codewords required to resynchronize, therefore it takes values $X = 1, 2, \ldots$. The probability of resynchronizing in X codewords is denoted by the X–step state transition probability $P_X(I \rightarrow S)$. Thus the expectation $E(X)$:

$$E(X) = \sum_{X=1}^{\infty} (P_X(I \rightarrow S) \times X)$$

is the expected number of codewords, of some variable length code C, required to resynchronize when synchronization is lost. The expectation $E(X)$ is termed the *error recovery* value.

There are at least two models available to construct the error recovery of a variable length code C. The objective of each model is to construct and define the n–step state transition probabilities $P_n(I \rightarrow S)$ for the expectation formula $E(X)$. The two models considered here are: Maxted and Robinson [10] and Takishima *et al.* [13].

4.2 The Maxted and Robinson Model.

The Maxted and Robinson model [10] is a Markov model that involves constructing one–step state transition probabilities, that are used to construct a state transition diagram G. The error recovery value is then calculated by applying graph reduction methods to G.

4.2.1 Transmission Errors Specified.

The type of transmission error considered is a single bit inversion applied to a single codeword. For example let $\bar{c} = b_1 b_2 ... b_n$ be a codeword of length n. When $\bar{c}$ is transmitted and subject to a transmission error, one bit inversion in $\bar{c}$ occurs. For codeword $\bar{c}$ of length n there are n combinations:

$$b_1' b_2 \ldots b_n;\ b_1 b_2' \ldots b_n;\ \ldots\ ;\ b_1 b_2 \ldots b_{n-1} b_n'$$

where b_i' denotes the ith bit of $\bar{c}$ in error.

4.2.2 Constructing One–Step State Transition Probabilities.

Construction of one–step state transition probabilities require the formation of tables; for each state a table is constructed. For example, let $C = \{0, 10, 11\}$ be a variable length code with $F = \{1\}$ the set of prefix words of C. The transmission probabilities of codewords in C are defined in Table 4.1:

Transmission Probability	C
0.5	0
0.25	10
0.25	11

Table 4.1: Transmission probabilities of a Huffman code C.

with an average codeword length $V = 1.5$. To construct one–step state transition probabilities from initial state I, form Table 4.2. In Table 4.2 each codeword is subject to one bit inversion; this includes all combinations of bit errors for each codeword. Consider codeword 11 with transmission probability $p_4 = 0.25$. The first bit of 11 is inverted to produce the string 01. Using a decoder D to decode 01, the initial state of D is I as D reads the first bit of 01. As D reads the last bit of 01

the resultant state of D is 1 because 0 is a codeword and 1 is a prefix word. Thus the state of D changes from I to 1 after decoding 01. The probability of this is deemed to be: $\frac{0.25}{1.5}$; the reception probability of codeword 11 divided by the average codeword length of C. Similarly, other rows in Table 4.2 are constructed.

i	Codeword Probability (p_i)	Codeword	Bit Inverted	Codeword In Error	Resultant State	Reception Probability (p_i/V)
1	0.5	0	1	1	1	0.5/1.5
2	0.25	10	1	00	S	0.25/1.5
3	0.25	10	2	11	S	0.25/1.5
4	0.25	11	1	01	1	0.25/1.5
5	0.25	11	2	10	S	0.25/1.5

Table 4.2: Construction of one–step state transition probabilities from state I.

Using Table 4.2, one–step state transition probabilities from initial state I are formed:

$$P(I \to S) = \frac{p_2}{V} + \frac{p_3}{V} + \frac{p_5}{V}$$

that is:

$$P(I \to S) = \frac{0.25}{1.5} + \frac{0.25}{1.5} + \frac{0.25}{1.5} = \frac{1}{2}$$

and

$$P(I \to 1) = \frac{p_1}{V} + \frac{p_4}{V}$$

that is:

$$P(I \to 1) = \frac{0.5}{1.5} + \frac{0.25}{1.5} = \frac{1}{2}$$

To construct one–step state transition probabilities from the error state 1, form Table 4.3. In Table 4.3, consider codeword 10, this is appended to the error state 1 to form the string 110. As D reads the first bit of 110 the state of D is 1, then after reading the last bit of 110 the state of D becomes S because 110 is a string of two codewords. Hence the probability of D changing from

error state 1 to synchronized state S, after reading codeword 10, equals the transmission probability of 10: 0.25. Similarly, other rows in Table 4.3 are constructed.

i	Error State	Codeword	Codeword Probability	Error State⊙Codeword	Resultant State	Reception Probability
1	1	0	0.5	10	S	$0.5=p_1$
2	1	10	0.25	110	S	$0.25=p_2$
3	1	11	0.25	111	1	$0.25=p_3$

Table 4.3: Construction of one-step state transition probabilities from error state 1.

Using Table 4.3, one-step state transition probabilities from error state 1 are formed:

$$P(1 \rightarrow S) = p_2 + p_1$$

that is:

$$P(1 \rightarrow S) = 0.5 + 0.25 = 0.75$$

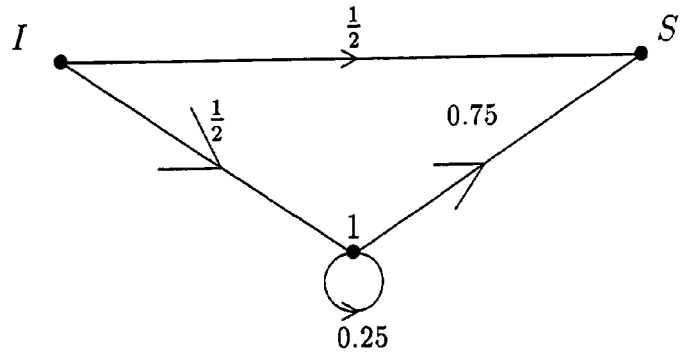
and

$$P(1 \rightarrow 1) = p_3$$

that is:

$$P(1 \rightarrow 1) = 0.25$$

After constructing one-step state transition probabilities, form a state transition diagram:



The diagram can be used to derive the error recovery value by means of graph reduction methods.

4.2.3 Graph Reduction Methods.

Graph reduction methods [10] are used to obtain n–step state transition probabilities $P_n(I \rightarrow S)$ from a state transition diagram G. The methods transform G into a generating function $G(z)$, termed the *gain* of G, that is used to derive the error recovery value.

A selection of four basic graph reduction/transformation rules are stated here, each producing the gain $G(z)$. Observe that in all state diagrams, it is necessary to multiply each transition probability by z in order to construct a generating polynomial $G(z)$.

1. In Figure 4.2, no graph reduction methods need to be applied, this is the required form:

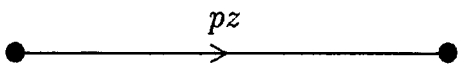


Figure 4.2: One path from source to sink.

hence the gain is:

$$G(z) = pz$$

2. In Figure 4.3 the probability of synchronizing is pz *or* qz:

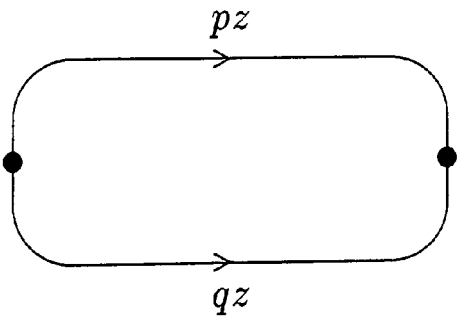


Figure 4.3: Two paths from source to sink.

hence the total gain is the sum of pz and qz:

$$G(z) = pz + qz = (p+q)z$$

which reduces the graph to that in Figure 4.4:

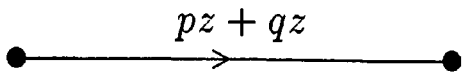


Figure 4.4: Total gain from source to sink.

3. In Figure 4.5 the probability of synchronizing is pz *and* qz:

Figure 4.5: One path from source to sink.

hence the total gain is the product pz and qz:

$$G(z) = pz \times qz = (pq)z^2$$

which reduces the graph to that in Figure 4.6:

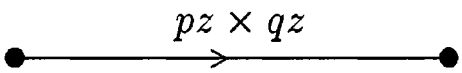


Figure 4.6: The gain from source to sink.

4. In Figure 4.7 the total gain is $G(z) = \frac{(pq)z^2}{1-rz}$:

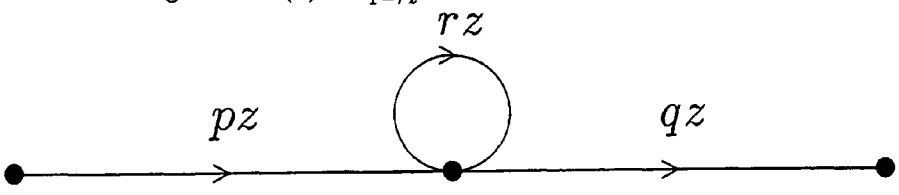


Figure 4.7: One path, in two or more steps, from source to sink.

To derive $G(z)$ in this case, observe that there are two or more paths from the source to the sink. Informally, the gain is:

$$\left(pz \times qz \right) \text{ or } \left(pz \times rz \times qz \right) \text{ or } \left(pz \times rz \times rz \times qz \right) \text{ or } \ldots \text{ or } \left(pz \times rz \ldots \times rz \times qz \right)$$

that is:

$$(pq)z^2 \times \sum_{i=0}^{\infty}(rz)^i$$

Using

$$\lim_{n\to\infty}\sum_{i=0}^{n}(x)^i = \frac{1}{1-x}$$

then

$$G(z) = \frac{(pq)z^2}{1-rz}$$

and the graph in Figure 4.7 reduces to that in Figure 4.8:

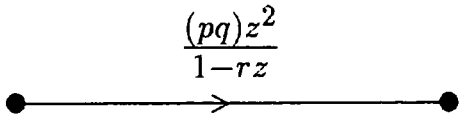
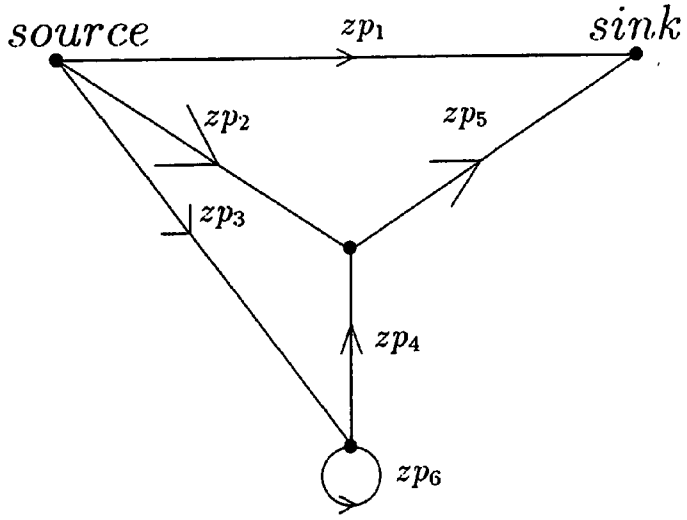


Figure 4.8: The total gain from source to sink.

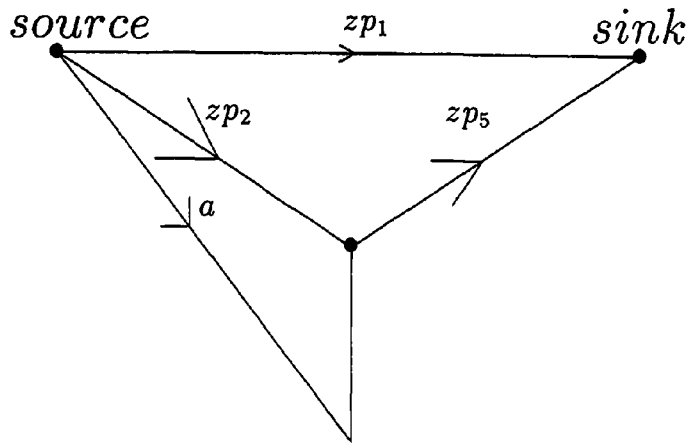
As an example of applying graph reduction transformations to obtain a generator polynomial, let G denote the state transition diagram:



labelled with state transition probabilities $p_1, p_2, p_3, p_4, p_5, p_6$, each multiplied by the factor z.

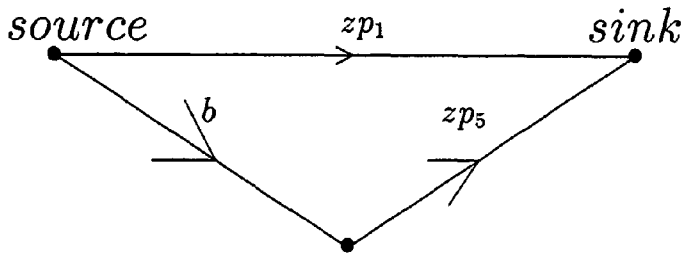
In steps, apply transformations to G:

- apply the transformation rule number (4.):



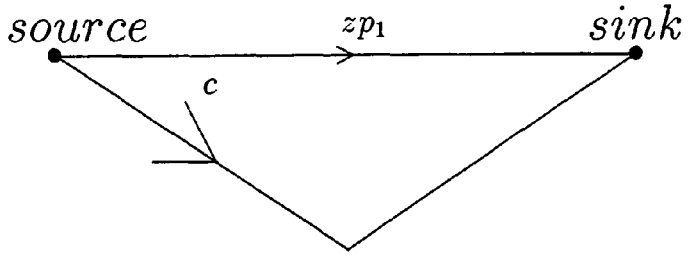
with $a = (zp_3)(zp_4)/(1 - zp_6)$;

- apply the transformation rule number (2.):



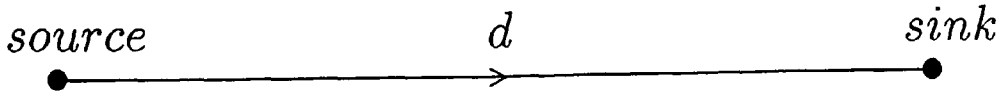
with $b = a + zp_2$;

- apply the transformation rule number (3.):



with $c = b \times zp_5$;

- apply the transformation rule number (2.):



with $d = c + zp_1$.

The total gain d equals $G(z)$:

$$G(z) = d$$

$$= c + zp_1$$

$$= (b \times zp_5) + zp_1$$

$$= (a + zp_2) \times zp_5 + zp_1$$

$$= (\tfrac{(zp_3)(zp_4)}{(1-zp_6)} + zp_2) \times zp_5 + zp_1$$

$$= \left(\tfrac{z^2p_3p_4}{(1-zp_6)} + zp_2\right) \times zp_5 + zp_1$$

thus the generator polynomial is obtained:

$$G(z) = \tfrac{z^3p_3p_4p_5}{(1-zp_6)} + z^2p_2p_5 + zp_1$$

$$= (1 + zp_6 + z^2p_6^2 + z^3p_6^3 + \ldots)(z^3p_3p_4p_5) + (z^2p_2p_5 + zp_1)$$

$$= zp_1 + z^2p_2p_5 + z^3p_3p_4p_5 + z^4p_3p_4p_5p_6$$
$$+z^5p_3p_4p_5p_6^2 + z^6p_3p_4p_5p_6^3 + \ldots$$

4.2.4 Graph Reduction Method: Mason's Gain Formula.

Mason's gain formula [9] is used to calculate the gain of a graph. By multiplying all transition probabilities in G by z then applying Mason's gain formula, a generator polynomial $G(z)$ in z is formed. The form of $G(z)$ will be written as a fraction because Mason's formula involves constructing a numerator and a denominator.

As an example of applying Mason's gain formula, form the state transition diagram, Figure 4.9, with transition probabilities: $s_1, s_2, t_1, t_2, r_1, r_2, a, b, c$. Then, form G in Figure 4.10 by multiplying all transition probabilities in Figure 4.9 by z.

Four steps are applied in sequence to G:

1. determining circuits and paths in G;

2. calculating the denominator;

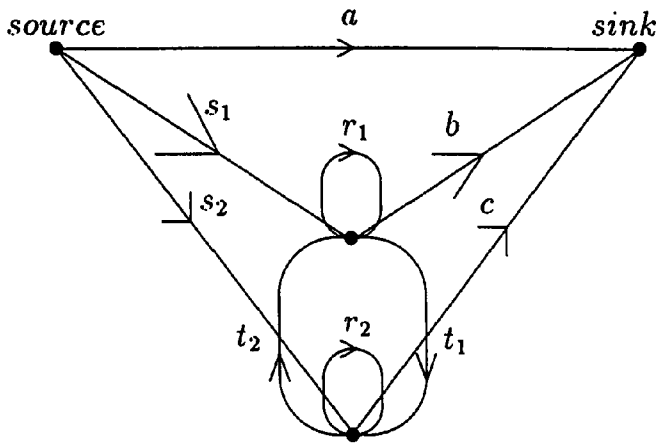


Figure 4.9: State transition diagram.

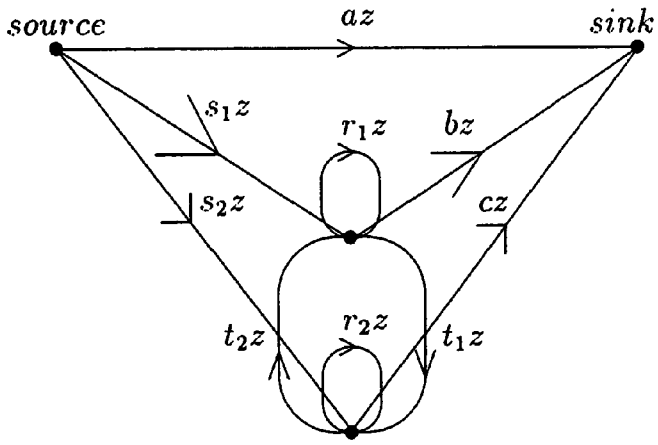


Figure 4.10: State transition diagram G.

3. calculating the numerator;

4. the total gain.

Apply the four steps to G:

1. A path is a sequence of edges starting from the source node and ending at the sink node; no edges or nodes are traversed more than once. In G the individual paths are:

$$G_1 = az, G_2 = s_1bz^2, G_3 = s_1t_1cz^3, G_4 = s_2cz^2, G_5 = s_2t_2bz^3$$

each one termed a *path gain*.

A circuit is similar to a path, that starts from a node n that is not equal to the source or sink node and ends on the original node n. Additionally, to form a circuit no edges or nodes are traversed more than once. In G, the individual circuits are:

$$T_1 = r_1 z, T_2 = r_2 z, T_3 = t_1 t_2 z^2$$

Disjoint circuits are two or more circuits in G that do not intersect: the nodes and edges in a circuit are distinct to those in another circuit. Similarly, when a path and a circuit do not intersect, they are also disjoint. The circuits $r_1 z$ and $r_2 z$ in G are disjoint, but circuits $r_1 z$ and $t_1 t_2 z^2$ are not, and similarly the path $s_2 c z^2$ is disjoint to circuit $r_1 z$ but not to circuit $r_2 z$.

2. Informally, the denominator is expressed as:

$$1 - \text{the sum of combinations of disjoint circuits}$$

The first sum of single circuits is: $Sum_{first} = T_1 + T_2 + T_3$, and the sum of pairs of disjoint circuits is: $Sum_{pairs} = T_1 T_2$, because T_1 and T_2 are the only disjoint pair. There are no more circuits to consider, hence there are no disjoint triple circuits for example.

The denominator equals the sum (A):

$$1 - Sum_{first} + Sum_{pairs}$$

observing that the sign in the sum alternates between minus and plus in each sum of odd and even products of disjoint circuits, respectively. That is:

$$1 - T_1 - T_2 - T_3 + T_1 T_2$$

3. The numerator is a sum of products, each product equals the path gain G_i times a factor Δ_i:

$$G_1 \, \Delta_1 + G_2 \, \Delta_2 + G_3 \, \Delta_3 + G_4 \, \Delta_4 + G_5 \Delta_5$$

Calculating the factor Δ_i is similar to calculating the denominator of Mason's formula:

$\Delta_i = 1 - \text{the sum of combinations of disjoint circuits, that are also disjoint to path } G_i$

with the sign alternating between minus and plus in each sum of odd and even products of disjoint circuits, respectively.

Calculate Δ_i for $i = 1 \ldots 5$:

- path G_1 : path $G_1 = az$ is disjoint to the individual circuits T_1, T_2 and T_3. The product of disjoint circuits T_1T_2 is also disjoint to the path G_1. Hence:

$$\Delta_1 = 1 - (T_1 + T_2 + T_3) + (T_1T_2)$$

  In this case Δ_1 equals the denominator (A).

- path G_2: there is one disjoint circuit to G_2: T_2, and consequently no multiples of disjoint circuits to G_2. Hence:

$$\Delta_2 = 1 - T_2$$

- path G_3: the path G_3 is in contact with all nodes in G, hence there are no disjoint circuits to G_3:

$$\Delta_3 = 1$$

- path G_4: there is one disjoint circuit to G_4: T_1, and consequently no multiples of disjoint circuits to G_4. Hence:

$$\Delta_4 = 1 - T_1$$

- path G_5: the path G_5 is in contact with all nodes in G, hence there are no disjoint circuits to G_5:

$$\Delta_5 = 1$$

The numerator in Mason's formula equals (B):

$$G_1(1 - (T_1 + T_2 + T_3) + (T_1T_2)) + G_2(1 - T_2) + G_3(1) + G_4(1 - T_1) + G_5(1)$$

4. The total gain is thus $(B)/(A)$:

$$\frac{G_1(1 - (T_1 + T_2 + T_3) + (T_1T_2)) + G_2(1 - T_2) + G_3(1) + G_4(1 - T_1) + G_5(1)}{(1 - (T_1 + T_2 + T_3) + (T_1T_2)}$$

4.2.5 Explanation of Mason's Gain Formula.

Mason's formula derives the generator polynomial

$$\frac{\sum_{i=1}^{k} G_i\Delta_i}{1 - H(z)}$$

from a state transition diagram G. Let $T_1, T_2, \ldots, T_N$ denote the individual circuits of G that are traversed on route from source to sink. The circuits $T_1, T_2, \ldots, T_N$ can be traversed more than once. For example, consider G from Section 4.2.4. The expression $s_1 r_1^2 b z^4$ is a route expressed as a four step sequence $s_1 z$, $r_1 z$, $r_1 z$, bz that traverses the circuit $r_1 z$ twice. Similarly $s_1 r_1^3 (t_1 t_2)^2 b z^9$ is another route from source to sink, traversing the circuits $r_1 z$ and $t_1 t_2 z^2$ more than once. In particular, the route az traverses no circuits.

With many combinations of circuit traversals to consider in a state transition diagram, there will be many combinations of n–step state transition probabilities from source to sink. The term

$$\frac{1}{1 - H(z)}$$

in Mason's formula is the generator polynomial for all n–step circuit traversals. The polynomial can be obtained by expanding $(1 - H(z))^{-1}$ to produce the series $J(z)$:

$$J(z) = 1 + H(z) + H(z)^2 + H(z)^3 + \ldots$$

For example, the individual circuits of G in Section 4.2.4 are defined:

$$T_1 = r_1 z, T_2 = r_2 z, T_3 = t_1 t_2 z^2$$

These circuits form the denominator $1 - H(z)$:

$$1 - (T_1 + T_2 + T_3 - T_1 T_2)$$

Expanding $(1 - H(z))^{-1}$ by the binomial theorem produces the series $J(z)$:

$$J(z) = 1 + T_1 + T_2 + T_3 + T_1^2 + T_2^2 + T_3^2 + T_1 T_2 + 2T_1 T_3 + 2T_2 T_3 + \cdots$$

that is the generator polynomial consisting of all n–step circuit traversals in G:

$$1 + (r_1 + r_2)z + (t_1 t_2 + r_1^2 + r_2^2 + r_1 r_2)z^2 + \cdots$$

where '1' denotes the *null* circuit: the traversal of no circuits. The *inclusion-exclusion* [1] form of $H(z)$ ensures that each individual circuit combination is counted once only.

The polynomial $J(z)$ is used to construct other polynomials of circuit traversals, each one specific to a path G_i. Denote by $K_i(z)$ the generator polynomial that consists of all circuit transversals disjoint to the path G_i; so that $\Delta_i = 1 - K_i(z)$. Consider the product $J(z)\Delta_i$:

$$J(z)\Delta_i = J(z) - J(z)K_i(z)$$

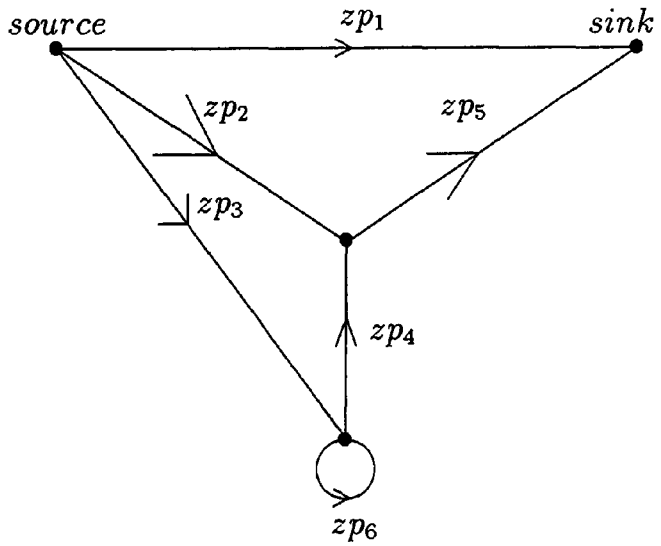
The polynomial $J(z)\Delta_i$ consists of all circuit transversals, excluding those that include a circuit that is not reachable from G_i. This is ensured by the inclusion–exclusion form of $H(z)$. For example, given path $G_2 = s_1 bz^2$ and $\Delta_2 = 1 - T_2$, the polynomial $J(z)\Delta_2$ consists of circuits that are reachable from G_2:

$$J(z)\Delta_2 = 1+T_1+T_3+T_1^2+T_3^2+2T_1T_3+3T_1^2T_3+T_3T_2+2T_1T_2T_3+T_2^2T_3+3T_1T_3^2+2T_2T_3^2+T_1^3+T_3^3\ldots.$$

For instance, consider the term T_3T_2, this implies that the route $G_2T_3T_2$ (that is $s_1t_1r_2t_2bz^5$) traverses the circuits T_3 and T_2. However, in $J(z)\Delta_2$ the term T_2^n, for $n \geq 1$, is not present. This is because T_2^n is not reachable from G_2: $G_2T_2^n$ is not a route from source to sink.

4.2.6 Example of Mason's Formula.

As an example of applying Mason's gain formula to construct a generating function from a state transition diagram G, let G denote the diagram:



with transition probabilities $p_1, p_2, p_3, p_4, p_5, p_6$ multiplied by the factor z. Apply the four steps of Mason's gain formula:

1. In G the individual circuits are: $T_1 = zp_6$, and individual paths are:

$$G_1 = zp_1, G_2 = zp_2zp_5, G_3 = zp_3zp_4zp_5$$

2. The denominator (A) equals:

$$1 - T_1$$

3. For each path G_i calculate Δ_i for $i = 1 \ldots 3$:

   (a) path G_1: there is one disjoint circuit to G_1: zp_6, and no others, hence $\Delta_1 = 1 - zp_6$.

   (b) path G_2: there is one disjoint circuit to G_2: zp_6, and no others, hence $\Delta_2 = 1 - zp_6$.

   (c) path G_3: the path traverses all nodes in G, hence there are no disjoint circuits to G_3: $\Delta_3 = 1$.

   The numerator (B) of Mason's gain formula equals:

$$G_1\,\Delta_1 + G_2\,\Delta_2 + G_3\Delta_3$$

4. The total gain equals $(B)/(A)$, that is:

$$\begin{aligned}
G(z) &= \tfrac{G_1(1-zp_6)+G_2(1-zp_6)+G_3}{1-zp_6} \\
&= \tfrac{zp_1(1-zp_6)+z^2p_2p_5(1-zp_6)+z^3p_3p_4p_5}{1-zp_6} \\
&= p_1z + p_2p_5z^2 + \left(\tfrac{p_3p_4p_5z^3}{1-zp_6}\right) \\
&= p_1z + p_2p_5z^2 + (1 + p_6z + p_6^2z^2 + p_6^3z^3 + \ldots)(p_3p_4p_5z^3) \\
&= p_1z + p_2p_5z^2 + p_3p_4p_5z^3 + p_3p_4p_5p_6z^4 \\
&\quad + p_3p_4p_5p_6^2z^5 + p_3p_4p_5p_6^3z^6 + \ldots
\end{aligned}$$

Observe that the generating function is equal to $G(z)$ in Section 4.2.3 that was obtained from the graph reduction method example: Mason's formula or graph reduction methods will produce the same generating function for the same graph G.

4.2.7 The Error Recovery from Graph Reduction Methods.

One–step state transition probabilities of a variable length code C are used to construct a state transition diagram G. Graph reduction methods are applied to G to construct the generating

function $G(z)$. To derive the error recovery value from $G(z)$, differentiate $G(z)$ with respect to z, then let $z = 1$. Thus $G'(1)$ is the theoretical error recovery value for C.

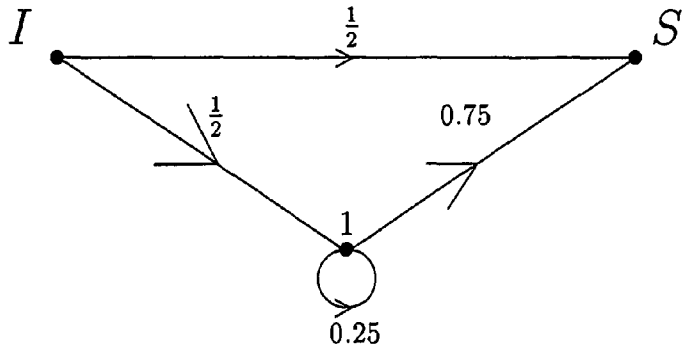
It can be shown that $G'(1)$ equals the expectation of a discrete random variable. Let $G(z) = p_1 z^1 + p_2 z^2 + p_3 z^3 + \ldots + p_n z^n$. The coefficient p_n of the nth term $p_n z^n$ is the transition probability $P_n(I \rightarrow S)$ of synchronizing in exactly n words following a transmission error. Calculating the expectation $E(X)$ of synchronizing, by using the coefficients of each term in $G(z)$, gives:

$$(1 \times p_1) + (2 \times p_2) + (3 \times p_3) + \ldots + (n \times p_n)$$

that is equivalent to the derivative of $G(z)$ with $z = 1$:

$$(1 \times p_1)z^0 + (2 \times p_2)z^1 + (3 \times p_3)z^2 + \ldots + (n \times p_n)z^{n-1}$$

For example, the code C first seen on page 56 in Table 4.1 was used to form the state transition diagram:



Applying graph reduction methods to this graph will produce the generating function:

$$G(z) = \frac{(1/2)(0.75)}{1 - 0.25z} z^2 + \frac{1}{2} z$$

Thus the error recovery of C, as predicted by the Maxted and Robinson model, is:

$$G'(1) = 1.67$$

4.3 The Takishima, Wada, Murakami Model.

The Takishima, Wada, Murakami error recovery model [13] involves constructing one-step state transition probabilities that are used to form a transition matrix M. The error recovery is then calculated by substituting M into a formula derived in [13].

4.3.1 Transmission Errors and String Reception Probabilities.

For each codeword from a variable length code C, *combinations* of bit inversion errors are applied. For example if the string 000 of length 3 is a codeword subject to errors, there are seven possibilities. Table 4.4 lists all possible bit inversion errors that can be applied to codeword 000:

Codeword	Inversion Bit	String Received
000	1	100
000	2	010
000	3	001
000	1+2	110
000	1+3	101
000	2+3	011
000	1+2+3	111

Table 4.4: Reception possibilities of a codeword in error.

In the presence of channel errors, each transmitted codeword $c \in C$ has an associated set of reception probabilities for the strings that may be received when c is transmitted. Consider the reception error probabilities associated with some codeword $\bar{c}$ of length n with transmission probability p. There are

$$\binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{n}$$

possibilities to consider if $\bar{c}$ is received in error. For each received string, its reception probability is calculated by the formula:

$$p \times (1 - P_{err})^{n-l}(P_{err})^{l}$$

where P_{err} denotes the probability of one bit inversion and $l \geq 0$ denotes the number of bit inversions.

For example, if the transmission probability of 000 was 0.5 and the probability P_{err} of one bit inversion error occuring to a codeword during transmission was 0.1, then there are eight reception probabilities to calculate: the probability that 000 is received error free and the seven possiblities that 000 is corrupted. Each error reception probability for 000 is calculated from the formula:

$$0.5 \times (1 - P_{err})^{3-l} \times (P_{err})^{l}$$

and is listed in Table 4.5:

Codeword Probability	Codeword	Bit Inverted	String Received	Reception Probability
0.5	000	1	100	$0.5 \times (0.9)^{3-1} \times (0.1)^{1} = 0.0405$
0.5	000	2	010	$0.5 \times (0.9)^{3-1} \times (0.1)^{1} = 0.0405$
0.5	000	3	001	$0.5 \times (0.9)^{3-1} \times (0.1)^{1} = 0.0405$
0.5	000	1+2	110	$0.5 \times (0.9)^{3-2} \times (0.1)^{2} = 0.0045$
0.5	000	1+3	101	$0.5 \times (0.9)^{3-2} \times (0.1)^{2} = 0.0045$
0.5	000	2+3	011	$0.5 \times (0.9)^{3-2} \times (0.1)^{2} = 0.0045$
0.5	000	1+2+3	111	$0.5 \times (0.9)^{3-3} \times (0.1)^{3} = 0.0005$

Table 4.5: String reception probabilities of one codeword in error.

with the reception probability that 000 is received error free, stated in Table 4.6:

Codeword Probability	Codeword	Bit Inverted	String Received	Reception Probability
0.5	000	0	000	$0.5 \times (0.9)^{3-0} \times (0.1)^{0} = 0.3645$

Table 4.6: String reception probability of a codeword received error free.

The reception probabilities of codewords that are in error and error free are used to construct one–step state transition probabilities. For each codeword $c \in C$ that is transmitted, the string received and its reception probability is used to construct the transition matrix M. The method of constructing M from received strings and reception probabilities is illustrated by using tables in Section 4.3.2.

4.3.2 Example of Constructing a Transition Matrix.

Let C be the variable length code first seen on page 56 in Table 4.1, that is repeated here in Table 4.7:

Probability	C
0.5	0
0.25	10
0.25	11

Table 4.7: Codewords and their transmission probabilities.

The set F of prefix words of C is $F = \{1\}$. F contains one element that will form the error state of the decoder D. Four tables are constructed. Table 4.8 gives reception probabilities for codewords in error:

Codeword Probability	Original Codeword	Bit Inverted	String Received	String Reception Probability
0.5	0	1	1	$0.5 \times (0.9)^0 \times (0.1)^1 = 0.05$
0.25	10	1	00	$0.25 \times (0.9)^{2-1} \times (0.1)^1 = 0.0225$
0.25	10	2	11	$0.25 \times (0.9)^{2-1} \times (0.1)^1 = 0.0225$
0.25	10	1+2	01	$0.25 \times (0.9)^{2-2} \times (0.1)^2 = 0.0025$
0.25	11	1	01	$0.25 \times (0.9)^{2-1} \times (0.1)^1 = 0.0225$
0.25	11	2	10	$0.25 \times (0.9)^{2-1} \times (0.1)^1 = 0.0225$
0.25	11	1+2	00	$0.25 \times (0.9)^{2-2} \times (0.1)^2 = 0.0025$
				$Total = 0.145$

Table 4.8: String reception probabilities of codewords in error.

Consider the first row of Table 4.8. The codeword 0 with transmission probability 0.5 is subject to one bit inversion, resulting in the string 1. The probability of receiving the string 1 is calculated: $0.5 \times (0.9)^0 \times (0.1)^1$ to give 0.05. Other rows in Table 4.8 are obtained in a similar way.

The second table, Table 4.9, is constructed as Table 4.8, except that no errors are applied to codewords. In Table 4.9 the probabilities of receiving each codeword error free are tabulated. The sum of all reception probabilities in Table 4.9 will equal P_{corr}, that is the probability of receiving an error free codeword:

$$P_{corr} = 0.45 + 0.2025 + 0.2025 = 0.855$$

Codeword Probability	Original Codeword	Bit Inverted	String Received	String Reception Probability
0.5	0	-	0	$0.5 \times (0.9)^1 \times (0.1)^0 = 0.45$
0.25	10	-	10	$0.25 \times (0.9)^2 \times (0.1)^0 = 0.2025$
0.25	11	-	11	$0.25 \times (0.9)^2 \times (0.1)^0 = 0.2025$

Table 4.9: Reception probabilities of codewords that are error free.

and the *normalization* of P_{corr} defines the probability P_{norm} of receiving a codeword that is not error free, hence $P_{norm} = 1 - P_{corr}$ equals the sum of string reception probabilities in Table 4.8:

$$P_{norm} = 1 - P_{corr} = 0.145$$

The columns in Table 4.8 ('*String Received*' and '*String Reception Probability*') are used to construct Table 4.10, (where λ denotes the null string):

String Received	λ	1	String Received Probability
0	S	S	0.45
1	1	S	0.05
00	S	S	0.025
01	1	1	0.025
10	S	S	0.225
11	S	1	0.225

Table 4.10: String reception probabilities and state transitions.

In the first column of Table 4.10, all possible strings received are listed, and for each string the reception probability is calculated, to form the fourth column. To calculate the string reception probability of 0, observe that in Table 4.9 (column '*String Received*') the string 0 occurs once with reception probability 0.45. To calculate the string reception probability of 00, the string 00 occurs twice in Table 4.8 with reception probability 0.0225 or 0.0025: hence the probability of receiving 00 equals the sum 0.0225+0.0025, to give 0.025.

To construct columns 2 and 3 in Table 4.10, each string in column 1 is appended to state λ and

error state 1, then decoded, and the resulting state is noted. For example, in column λ, append each string and decode:

String Received	λ
0	$\lambda \odot 0 \equiv S$
1	$\lambda \odot 1 \equiv 1$
00	$\lambda \odot 00 \equiv S$
01	$\lambda \odot 01 \equiv 1$
10	$\lambda \odot 10 \equiv S$
11	$\lambda \odot 11 \equiv S$

The last three columns ('λ','1' and '*String Received Probability*') in Table 4.10 are used to construct the rows of Table 4.11. Using Table 4.10, transition probabilities $P(I \rightarrow S)$ and $P(I \rightarrow 1)$ from state I are obtained from column 'λ': for each occurence of S in column 'λ', sum the corresponding reception probabilities (in the adjacent column) to give the total $P(I \rightarrow S)$:

$$P(I \rightarrow S) = 0.45 + 0.025 + 0.225 + 0.225 = 0.925$$

and similarly for each occurence of 1 in column 'λ', sum the corresponding reception probabilities to give the total $P(I \rightarrow 1)$:

$$P(I \rightarrow 1) = 0.05 + 0.025 = 0.075$$

Repeating this procedure to column '1' of Table 4.10, transition probabilities $P(1 \rightarrow S)$ and $P(1 \rightarrow 1)$ from state 1 are obtained, thus completing the construction of Table 4.11:

	S	1
I	0.925	0.075
1	0.75	0.25

Table 4.11: The state transition table.

To obtain the matrix M from Table 4.11, we must subtract P_{corr} from the entry $P(I \rightarrow S)$, this is because each element in the sum P_{corr} is a string reception probability that a codeword $c \in C$ is received error free. If c is a codeword received error free, the initial state of decoder D is undefined

because c contains no errors: as D reads the first bit of c, the state is not I. Consequently, after subtracting P_{corr}, we must normalise the first row of Table 4.11 by dividing all entries by P_{norm}; for the sum of entries to equal 1. Thus the the transition matrix M defined by [13] for the variable length code C is:

$$M = \begin{pmatrix} \frac{0.925-0.855}{0.145} & \frac{0.075}{0.145} \\ 0.75 & 0.25 \end{pmatrix} = \begin{pmatrix} 0.48 & 0.52 \\ 0.75 & 0.25 \end{pmatrix}$$

4.3.3 The Error Recovery Formula E_{rec}.

When the matrix M is known the error recovery value E_{rec} is obtained by substituting submatrices of M into the formula

$$E_{rec} = p_{(0,0)} + Q(1)\left\{(I-T)^{-1} + (I-T)^{-2}\right\}R$$

that was derived by Takishima *et al.* in [13]. In E_{rec}, $Q(1)$ and R are the row and column vectors of a transition matrix M, where $q_{(0,i)}$ and $s_{(i,0)}$ are defined as in Section 4.1.4:

$$Q(1) = (q_{(0,1)}, q_{(0,2)}, \cdots, q_{(0,N)})$$

$$R = \begin{pmatrix} s_{(1,0)} \\ s_{(2,0)} \\ \vdots \\ s_{(N,0)} \end{pmatrix}$$

with T the matrix of error state transition probabilities:

$$T = \begin{pmatrix} t_{(1,1)} & t_{(1,2)} & \cdots & t_{(1,N)} \\ t_{(2,1)} & t_{(2,2)} & \cdots & t_{(2,N)} \\ \vdots & \vdots & \vdots & \vdots \\ t_{(N,1)} & t_{(N,2)} & \cdots & t_{(N,N)} \end{pmatrix}$$

and I denoting the $N \times N$ identity matrix for multiplication:

$$I = I_{N \times N}$$

The use of E_{rec} will be illustrated in the following example. The transition matrix M

$$M = \begin{pmatrix} 0.48 & 0.52 \\ 0.75 & 0.25 \end{pmatrix}$$

was derived from the Huffman code C first seen on page 56 in Table 4.1, and previously used on page 73 in Table 4.7, is repeated here in Table 4.12:

Probability	C
0.5	0
0.25	10
0.25	11

Table 4.12: Codeword transmission probabilities.

Using the error recovery formula E_{rec}, a theoretical value for the expected synchronization delay is obtained. To substitute into the E_{rec} formula, form the necessary matrices:

$$\begin{aligned} \mathrm{Q}(1) &= [0.52] \\ \mathrm{R} &= [0.75] \\ \mathrm{T} &= [0.25] \end{aligned}$$

and with $p_{(0,0)} = 0.48$:

$$E_{rec} = [0.48] + [0.52] \left\{ (1 - 0.25)^{-1} + (1 - 0.25)^{-2} \right\} [0.75] = [1.693]$$

Hence when a transmission error occurs to a bit stream composed of the codewords from C we expect on average that synchronization is regained in 1.693 codewords.

4.3.4 Derivation of the Error Recovery Formula E_{rec}.

Deriving the error recovery formula E_{rec} involves constructing an expression for the n–step state transition probability $P_n(I \rightarrow S)$, to be substituted into the expectation formula $E(X)$:

$$E(X) = \sum_{X=1}^{\infty} (X \times P_X(I \rightarrow S))$$

then calculating the limiting value of $E(X)$.

First we prove a preliminary result.

Proposition 1 *Assuming that the code is able to recover synchronization*

$$\lim_{n \rightarrow \infty} n^{\alpha} T^n = 0$$

for $\alpha \in \{0, 1, 2\}$.

Proof.

Here we will prove the case $\alpha = 2$. The cases $\alpha = 0, 1$ are similar and are proved in [13]. The proof by induction is divided into 7 stages:

1. Notation:

   (a) Denote the $N \times N$ matrix T by $T(N)$, for $N \geq 1$.

   (b) The nth power of T is denoted by $T^n(N)$, for $n \geq 1$.

   (c) The element in the ith row jth column of $T^n(N)$ is denoted by $t_{(i,j)}(N, n)$ with $1 \leq i, j \leq N$.

   (d) $T(N)$ is a submatrix of M:

$$M = \begin{pmatrix} p_{(0,0)} & q_{(0,1)} & q_{(0,2)} & \cdots & q_{(0,N)} \\ s_{(1,0)} & t_{(1,1)} & t_{(1,2)} & \cdots & t_{(1,N)} \\ s_{(2,0)} & t_{(2,1)} & t_{(2,2)} & \cdots & t_{(2,N)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ s_{(N,0)} & t_{(N,1)} & t_{(N,2)} & \cdots & t_{(N,N)} \end{pmatrix}$$

   It is assumed that $s_{(i,0)}$ in M cannot equal 0 at the same time for all $i = 1, \ldots, N$ in $T(N)$. The proof assumes that we are discussing a variable length code which does not include an infinite loop of error states:

$$\sum_{j=1}^{N} t_{(i,j)}(N,n) = 1 - s_{(j,0)} \leq 1$$

for all $i = 1, \ldots, N$. This means that for at least one error state, there exists at least one string, that is a codeword or a codeword in error, that will synchronize. This condition is also assumed in the proofs of $T^n = 0$ and $nT^n = 0$ as n tends to infinity.

2. When $N = 1$:

$$[t_{(1,1)}(1,1)] = T(1)$$

then M is specified:

$$M = \begin{pmatrix} p_{(0,0)} & q_{(0,1)} \\ s_{(1,0)} & t_{(1,1)} \end{pmatrix}$$

Raising $T(1)$ to the power of n and multiplying by n^2 gives:

$$n^2 T^n(1) = n^2[t_{(1,1)}(1,n)] = n^2[t_{(1,1)}(1,1)]^n$$

With $t_{(1,1)}(1,1) < 1$ when $n = 1$, by statement 1.(d):

$$\lim_{n \to \infty} n^2 T^n(1) = 0$$

by a standard result from analysis.

3. Assume that

$$\lim_{n \to \infty} n^2 T^n(p) = 0$$

is true when $N = p$.

4. When $N = p + 1$:

Define the matrix $T_1(p+1)$:

$$T_1(p+1) = \begin{pmatrix} T_1(p) & & t_{(1,p+1)}(p+1,1) \\ & & \vdots \\ t_{(p+1,1)}(p+1,1) & \cdots & t_{(p+1,p+1)}(p+1,1) \end{pmatrix}$$

Consider the nth product $T_1^n(p+1)$, multiplied by n^2:

$$n^2T_1^n(p+1) = n^2 \begin{pmatrix} T_1^n(p) & & t_{(1,p+1)}(p+1,n) \\ & & \vdots \\ t_{(p+1,1)}(p+1,n) & \dots & t_{(p+1,p+1)}(p+1,n) \end{pmatrix}$$

with

$$\lim_{n\to\infty} n^2T_1^n(p) = 0$$

as assumed in part 3. Similarly define the matrix $T_2(p+1)$:

$$T_2(p+1) = \begin{pmatrix} t_{(1,1)}(p+1,1) & \dots & t_{(1,p+1)}(p+1,1) \\ \vdots & & \\ t_{(p+1,1)}(p+1,1) & & T_2(p) \end{pmatrix}$$

Consider the nth product $T_2^n(p+1)$, multiplied by n^2:

$$n^2T_2^n(p+1) = n^2 \begin{pmatrix} t_{(1,1)}(p+1,n) & \dots & t_{(1,p+1)}(p+1,n) \\ \vdots & & \\ t_{(p+1,1)}(p+1,n) & & T_2^n(p) \end{pmatrix}$$

with

$$\lim_{n\to\infty} n^2T_2^n(p) = 0.$$

We deduce that

$$\lim_{n\to\infty} n^2t_{(i,j)}(p+1,n) = 0$$

for all i,j in $1 \leq i,j \leq p+1$ except for $(i,j) = (p+1,1)$ and $(i,j) = (1,p+1)$, which are unknown. The remainder of this proof will show that both $n^2t_{(p+1,1)}(p+1,n)$ and $n^2t_{(1,p+1)}(p+1,n)$ will tend to zero as n tends to infinity.

5. Let $n^2T^n(p+1) = n^2T(p+1) \times T^{n-1}(p+1)$ then determine an expression for the element $n^2t_{(p+1,1)}(p+1,n)$:

$$n^2t_{(p+1,1)}(p+1,n) = n^2 \textstyle\sum_{k=1}^{p+1} \{t_{(p+1,k)}(p+1,1)\ t_{(k,1)}(p+1,n-1)\}$$

$$= n^2 \left\{t_{(p+1,p+1)}(p+1,1)\ t_{(p+1,1)}(p+1,n-1)\right\}$$

$$+n^2 \textstyle\sum_{k=1}^{p} \left\{t_{(p+1,k)}(p+1,1)\ t_{(k,1)}(p+1,n-1)\right\}$$

Let $\max(n-1)$ denote the maximum value in the first column of $T^{n-1}(p+1)$, excluding the element $t_{(p+1,1)}(p+1,n-1)$. That is $\max(n-1)$ denotes the maximum value of $t_{(k,1)}(p+1,n-1)$ for $1 \leq k \leq p$, thus forming an upper bound:

$$n^2 t_{(p+1,1)}(p+1,n) \leq n^2 \left\{ t_{(p+1,p+1)}(p+1,1) t_{(p+1,1)}(p+1,n-1) \right\}$$

$$+n^2 \{1 - t_{(p+1,p+1)}(p+1,1)\} \max(n-1)$$

Observe that:

$$\lim_{n \to \infty} n^2 \max(n-1) = 0.$$

because

$$\lim_{n \to \infty} n^2 t_{(i,j)}(p+1,n-1) = 0$$

for all i,j in $1 \leq i,j \leq p+1$ except for $(i,j) = (p+1,1)$ and $(i,j) = (1,p+1)$, as stated in part 4. We need to prove that the bound on $n^2 t_{(p+1,1)}(p+1,n)$ tends to 0 as n tends to infinity.

6. Consider the two cases, when $t_{(p+1,p+1)}(p+1,1) = 1$ and $t_{(p+1,p+1)}(p+1,1) < 1$:

   (a) If $t_{(p+1,p+1)}(p+1,1) = 1$ and $n \to \infty$:

   $$n^2 t_{(p+1,1)}(p+1,n) \leq n^2 t_{(p+1,1)}(p+1,n-1)$$

   then

   $$n^2 t_{(p+1,1)}(p+1,n-1) \leq n^2 t_{(p+1,1)}(p+1,n-2)$$
   $$\vdots$$
   $$n^2 t_{(p+1,1)}(p+1,2) \leq n^2 t_{(p+1,1)}(p+1,1)$$

   so that:

   $$n^2 t_{(p+1,1)}(p+1,n) \leq n^2 t_{(p+1,1)}(p+1,1)$$

   but $t_{(p+1,1)}(p+1,1) = 0$ if $t_{(p+1,p+1)}(p+1,1) = 1$, therefore $lim\ n^2 t_{(p+1,1)}(p+1,n) = 0$ for all n when $t_{(p+1,p+1)}(p+1,1) = 1$.

   (b) If $t_{(p+1,p+1)}(p+1,1) < 1$. A new bound on $n^2 t_{(p+1,1)}(p+1,n)$ is sought. As

   $$\lim_{n \to \infty} n^2 \max(n-1) = 0$$

   then

$$n^2 \max(n-1) < \epsilon$$

for all $n > m_0$ where m_0 is constant. With these constraints the new upper bound on $n^2 t_{(p+1,1)}(p+1,n)$ can be formed.

Given $n > m_0$ and since $m_0 + 1 > m_0$, we can obtain an expression for the element $n^2 t_{(p+1,1)}(p+1, m_0+1)$ from the matrix $n^2 T^{m_0+1}(p+1)$. That is, from the identity $n^2 T^{m_0+1}(p+1) = T(p+1) \times n^2 T^{m_0}(p+1)$:

$$\begin{aligned} n^2 t_{(p+1,1)}(p+1, m_0+1) &= n^2 \textstyle\sum_{k=1}^{p+1} \{t_{(p+1,k)}(p+1,1)\ t_{(k,1)}(p+1, m_0)\} \\ &= n^2 \left\{t_{(p+1,p+1)}(p+1,1) t_{(p+1,1)}(p+1, m_0)\right\} \\ &\quad + n^2 \textstyle\sum_{k=1}^{p} \left\{t_{(p+1,k)}(p+1,1) t_{(k,1)}(p+1, m_0)\right\} \end{aligned}$$

With $n^2 \max(n-1) < \epsilon$ a strict upper bound is formed:

$$\begin{aligned} n^2 t_{(p+1,1)}(p+1, m_0+1) &< n^2 \left\{t_{(p+1,p+1)}(p+1,1) t_{(p+1,1)}(p+1, m_0)\right\} \\ &\quad + \{1 - t_{(p+1,p+1)}(p+1,1)\}\epsilon \end{aligned}$$

and similarly, with this bound determine one for the element $n^2 t_{(p+1,1)}(p+1, m_0+2)$ in the matrix $n^2 T^{m_0+2}(p+1)$ by letting $n^2 T^{m_0+2}(p+1) = T(p+1) \times n^2 T^{m_0+1}(p+1)$:

$$\begin{aligned} n^2 t_{(p+1,1)}(p+1, m_0+2) &< t_{(p+1,p+1)}(p+1,1) \\ &\quad \times (n^2 \left\{t_{(p+1,p+1)}(p+1,1) t_{(p+1,1)}(p+1, m_0)\right\} \\ &\quad + \{1 - t_{(p+1,p+1)}(p+1,1)\}\epsilon) \\ &\quad + \{1 - t_{(p+1,p+1)}(p+1,1)\}\epsilon \\ &< n^2 \left[t_{(p+1,p+1)}(p+1,1)\right]^2 t_{(p+1,1)}(p+1, m_0) \\ &\quad + \left[t_{(p+1,p+1)}(p+1,1) + 1\right] \left\{1 - t_{(p+1,p+1)}(p+1,1)\right\} \epsilon \end{aligned}$$

and similarly determine a bound for $n^2 t_{(p+1,1)}(p+1, m_0+3)$:

$$n^2 t_{(p+1,1)}(p+1, m_0+3) < t_{(p+1,p+1)}(p+1,1)$$

$$\times (n^2 \left[t_{(p+1,p+1)}(p+1,1)\right]^2 t_{(p+1,1)}(p+1, m_0)$$

$$\left[t_{(p+1,p+1)}(p+1,1)+1\right]\left\{1 - t_{(p+1,p+1)}(p+1,1)\right\}\epsilon)$$

$$\left\{1 - t_{(p+1,p+1)}(p+1,1)\right\}\epsilon$$

$$< n^2 \left[t_{(p+1,p+1)}(p+1,1)\right]^3 t_{(p+1,1)}(p+1, m_0)$$

$$+ \left[\left\{t_{(p+1,p+1)}(p+1,1)\right\}^2 + t_{(p+1,p+1)}(p+1,1)+1\right]$$

$$\times \left\{1 - t_{(p+1,p+1)}(p+1,1)\right\}\epsilon$$

continuing this process an upper bound on $n^2 t_{(p+1,1)}(p+1, m_0+(n-m_0))$ is derived:

$$n^2 t_{(p+1,1)}(p+1,n) < n^2 \left[t_{(p+1,p+1)}(p+1,1)\right]^{n-m_0} t_{(p+1,1)}(p+1, m_0)$$

$$+ \left[\sum_{i=0}^{n-m_0-1}\left\{t_{(p+1,p+1)}(p+1,1)\right\}^i\right]\left\{1 - t_{(p+1,p+1)}(p+1,1)\right\}\epsilon$$

that simplifies to:

$$n^2 t_{(p+1,1)}(p+1,n) < n^2 \left[t_{(p+1,p+1)}(p+1,1)\right]^{n-m_0} t_{(p+1,1)}(p+1, m_0)$$

$$+ \left\{1 - \left[t_{(p+1,p+1)}(p+1,1)\right]^{n-m_0}\right\}\epsilon$$

With

$$\lim_{n\to\infty} n^2 \max(n-1) = 0$$

and

$$n^2 \max(n-1) < \epsilon$$

then ϵ can be made small so that

$$\lim_{n\to\infty \epsilon\to 0}\left\{1 - \left[t_{(p+1,p+1)}(p+1,1)\right]^{n-m_0}\right\}\epsilon = 0$$

Also with m_0 a constant and therefore $t_{(p+1,1)}(p+1, m_0)$ some constant, we can state that

$$\lim_{n\to\infty} n^2 \left[t_{(p+1,p+1)}(p+1,1)\right]^{n-m_0} t_{(p+1,1)}(p+1,m_0) = 0$$

when $t_{(p+1,p+1)}(p+1,1) < 1$.

Hence:

$$\lim_{n\to\infty} n^2 t_{(p+1,1)}(p+1,n) = 0$$

Using methods similar to those in (a) and (b) it can be proved that

$$\lim_{n\to\infty} n^2 t_{(1,p+1)}(p+1,n) = 0$$

This implies that

$$\lim_{n\to\infty} n^2 T^n(p+1) = 0$$

7. From

$$\lim_{n\to\infty} n^2 T^n(p) = 0$$

we have shown

$$\lim_{n\to\infty} n^2 T^n(p+1) = 0$$

Since

$$\lim_{n\to\infty} n^2 T^n(1) = 0$$

then

$$\lim_{n\to\infty} n^2 T^n(M) = 0$$

for all integer M by induction.

Thus

$$\lim_{n\to\infty} n^2 T^n = 0.$$

This completes the proof of proposition 1 for case $\alpha = 2$.

To form an expression for the nth term of $P_n(I \rightarrow S)$ let M denote the transition matrix of order $(N \times 1) \times (N \times 1)$, as defined in Section 4.1.4:

$$M = \begin{pmatrix} p_{(0,0)} & q_{(0,1)} & q_{(0,2)} & \cdots & q_{(0,N)} \\ s_{(1,0)} & t_{(1,1)} & t_{(1,2)} & \cdots & t_{(1,N)} \\ s_{(2,0)} & t_{(2,1)} & t_{(2,2)} & \cdots & t_{(2,N)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ s_{(N,0)} & t_{(N,1)} & t_{(N,2)} & \cdots & t_{(N,N)} \end{pmatrix}$$

and form the matrix T of order $N \times N$:

$$T = \begin{pmatrix} t_{(1,1)} & t_{(1,2)} & \cdots & t_{(1,N)} \\ t_{(2,1)} & t_{(2,2)} & \cdots & t_{(2,N)} \\ \vdots & \vdots & \vdots & \vdots \\ t_{(N,1)} & t_{(N,2)} & \cdots & t_{(N,N)} \end{pmatrix}$$

Denote by $p_i(n)$ the n–step transition probability from the initial state I to an error state i. With N error states there are N state transition probabilities to consider, therefore form a vector of n–step state transition probabilities:

$$(p_1(n), p_2(n), \ldots, p_N(n))$$

For example, the vector $(p_1(1), p_2(1), \ldots, p_N(1))$ equals the row vector $(q_{(0,1)}, q_{(0,2)}, \ldots, q_{(0,N)})$ in M:

$$(p_1(1), p_2(1), \ldots, p_N(1)) = (q_{(0,1)}, q_{(0,2)}, \ldots, q_{(0,N)})$$

To calculate the vector $(p_1(n), p_2(n), \ldots, p_N(n))$ for $n > 1$ form the vector recurrence formula using matrix T:

$$(p_1(n), p_2(n), \ldots, p_N(n)) = (p_1(n-1), p_2(n-1), \ldots, p_N(n-1)) \begin{pmatrix} t_{1,1} & t_{1,2} & \cdots & t_{1,N} \\ t_{2,1} & t_{2,2} & \cdots & t_{2,N} \\ \vdots & \vdots & \vdots & \vdots \\ t_{N,1} & t_{N,2} & \cdots & t_{N,N} \end{pmatrix}$$

which can be written as:

$$\wp(n) = \wp(n-1)T$$

for $n > 1$, with:

$$\wp(n) = (p_1(n), p_2(n), \ldots, p_N(n))$$

Using T and $Q(1) = (q_{(0,1)}, q_{(0,2)}, \ldots, q_{(0,N)})$ a simplified expression for $\wp(n)$ can be obtained. We have $\wp(n) = \wp(n-1)T$ and

$$\begin{aligned}
\wp(2) &= Q(1)T \\
\wp(3) &= \wp(2)T = (Q(1)T)T = Q(1)T^2 \\
\wp(4) &= \wp(3)T = (\wp(2)T)T = Q(1)T^3 \\
&\vdots \\
\wp(n) &= Q(1)T^{n-1}
\end{aligned}$$

The vector $\wp(n)$ of N n–step state transition probabilities is used to calculate the n–step state transition probability $P_n(I \rightarrow S)$ by using a column vector of M:

$$P_n(I \rightarrow S) = (p_1(n-1), p_2(n-1), \ldots, p_N(n-1)) \begin{pmatrix} s_{(1,0)} \\ s_{(2,0)} \\ \vdots \\ s_{(N,0)} \end{pmatrix}$$

i.e.

$$\begin{aligned}
P_n(I \rightarrow S) &= \wp(n-1)R \\
&= Q(1)T^{n-2}R
\end{aligned}$$

for $n \geq 2$, with

$$R = \begin{pmatrix} s_{(1,0)} \\ s_{(2,0)} \\ \vdots \\ s_{(N,0)} \end{pmatrix}.$$

Knowing that for $n = 1$, $P_1(I \rightarrow S) = p_{(0,0)}$, the expectation formula can be used:

$$E(X) = \sum_{X=1}^{\infty} X \times P_X(I \rightarrow S)$$

that is

$$E_{rec} = (1 \times p_{(0,0)}) + (2 \times Q(1)T^0R) + (3 \times Q(1)T^1R) + (4 \times Q(1)T^2R) + (5 \times Q(1)T^3R) + ..$$

with $T^0 = I_{N \times N}$ the identity matrix for multiplication.

To calculate the value E_{rec}, we must therefore determine the limit of the sum:

$$E_{rec} = \lim_{n \rightarrow \infty} \left(p_{(0,0)} + \sum_{i=1}^{n} (i+1)Q(1)T^{i-1}R \right)$$

that simplifies to:

$$E_{rec} = p_{(0,0)} + Q(1) \left(\lim_{n \rightarrow \infty} \sum_{i=1}^{n} (i+1)T^{i-1} \right) R$$

The limit of the sum

$$\sum_{i=1}^{n} (i+1)T^{i-1}$$

as $n \rightarrow \infty$ can be determined.

First calculate the summation $\sum_{i=1}^{n} T^{i-1}$. We have

$$(I - T)(I + T + \ldots + T^{n-1}) = I - T^n.$$

Multiplying both sides by $(I - T)^{-1}$ gives the expression

$$\sum_{i=1}^{n} T^{i-1} = (I - T^n)(I - T)^{-1}.$$

To calculate the summation $\sum_{i=1}^{n} iT^{i-1}$, let

$$A = I + 2T + 3T^2 + 4T^3 + \ldots + nT^{n-1}$$

and multiply both sides by $(I - T)$

$$\begin{aligned} A(I-T) &= I + T + T^2 + \ldots + T^{n-1} - nT^n \\ &= (I - T^n)(I-T)^{-1} - nT^n \end{aligned}$$

then the required expression for $\sum_{i=1}^{n} iT^{i-1}$ is obtained by multiplying both sides by $(I-T)^{-1}$

$$A(I-T)(I-T)^{-1} = ((I-T^n)(I-T)^{-1} - nT^n)(I-T)^{-1}$$

$$\sum_{i=1}^{n} iT^{i-1} = (I-T^n)(I-T)^{-2} - nT^n(I-T)^{-1}$$

Hence

$$\sum_{i=1}^{n}(i+1)T^{i-1} = (I-T^n)(I-T)^{-1} + (I-T^n)(I-T)^{-2} - nT^n(I-T)^{-1}$$

It was proved by Takishima *et al.* [13] that the matrix $(I-T)$ is non–singular. To see this, let

$$W(n) = I + T + \ldots + T^n$$

and multiply both sides by $(I-T)$ to give

$$(I-T)W(n) = I - T^{n+1}$$

From proposition 1,

$$\lim_{n\to\infty} T^n = \underline{0}$$

where $\underline{0}$ denotes the identity matrix under addition. Then

$$\lim_{n\to\infty}(I-T)W(n) = \lim_{n\to\infty}(I-T^{n+1}) = I.$$

Hence this shows that

$$(I-T)\lim_{n\to\infty} W(n) = I.$$

Since $(I-T)$ can have no column of all zeros, $W(n)$ converges to a finite limit W_0. Hence W is the inverse of $(I-T)$. So $(I-T)$ is non–singular.

Takishima *et al.* also proved the following expressions which were stated in proposition 1:

$$\lim_{n\to\infty} T^n = \underline{0}$$

and

$$\lim_{n\to\infty} nT^n = \underline{0}$$

Thus

$$\lim_{n\to\infty}\sum_{i=1}^{n}(i+1)T^{i-1} = (I-T)^{-1} + (I-T)^{-2}$$

therefore

$$E_{rec} = p_{(0,0)} + Q(1)\left((I-T)^{-1} + (I-T)^{-2}\right)R.$$

4.3.5 Derivation of the Variance Formula.

As well as knowing the value of E_{rec}, it would be useful to know the variance of this value. Here we develop an expression for the variance. In the variance formula

$$Var(X) = E(X^2) - E(X)^2$$

we know that $E_{rec} = E(X)$, therefore an expression for $E(X^2)$ in terms of the transition matrix T must be obtained. The value $E(X^2)$ has the following form:

$$E(X^2) = \sum_{X=1}^{\infty} X^2 \times P_X(I \to S)$$

$$E(X^2) = (1^2 \times p_{(0,0)}) + (2^2 \times Q(1)T^0R) + (3^2 \times Q(1)T^1R)) + (4^2 \times Q(1)T^2R) + \ldots$$

that is

$$E(X^2) = (1^2 \times p_{(0,0)}) + \lim_{n\to\infty} \left(\sum_{i=1}^{n} (i+1)^2 Q(1) T^{i-1} R \right)$$

$$= (1^2 \times p_{(0,0)}) + Q(1) \left(\lim_{n\to\infty} \sum_{i=1}^{n} (i+1)^2 T^{i-1} \right) R$$

Partitioning $E(X^2)$:

$$E(X^2) = p_{(0,0)} + Q(1) \left\{ \left[\lim_{n\to\infty} \sum_{i=1}^{n} i^2 T^{i-1} \right] + 2 \left[\lim_{n\to\infty} \sum_{i=1}^{n} i T^{i-1} \right] + \left[\lim_{n\to\infty} \sum_{i=1}^{n} T^{i-1} \right] \right\} R$$

determine the limit of each summand:

- $\lim_{n\to\infty} \sum_{i=1}^{n} T^{i-1}$: It was shown in the derivation of E_{rec} that

  $$\sum_{i=1}^{n} T^{i-1} = (I - T^n)(I - T)^{-1}$$

  therefore

  $$\lim_{n\to\infty} \sum_{i=1}^{n} T^{i-1} = (I - T)^{-1}$$

- $\lim_{n\to\infty} \sum_{i=1}^{n} iT^{i-1}$: It was noted in the derivation of E_{rec} that:

  $$\sum_{i=1}^{n} iT^{i-1} = (I - T^n)(I - T)^{-2} - nT^n(I - T)^{-1}$$

  hence

  $$\lim_{n\to\infty} \sum_{i=1}^{n} iT^{i-1} = (I - T)^{-2}$$

- $\lim_{n\to\infty} \sum_{i=1}^{n} i^2 T^{i-1}$:

Let

$$A = (1^2 I + 2^2 T + 3^2 T^2 + 4^2 T^3 + \ldots + (n-1)^2 T^{n-2} + n^2 T^{n-1})$$

and

$$B = (I - T)A$$

then

$$\begin{aligned} B &= (I + 2^2 T + 3^2 T^2 + 4^2 T^3 + \ldots + (n-1)^2 T^{n-2} + n^2 T^{n-1}) \\ &\quad -(T + 2^2 T^2 + 3^2 T^3 + \ldots + (n-1)^2 T^{n-1} + n^2 T^n) \\ &= I + (2^2 - 1^2)T + (3^2 - 2^2)T^2 + (4^2 - 3^2)T^3 + \ldots + (n^2 - (n-1)^2)T^{n-1} - n^2 T^n \\ &= (I + 3T + 5T^2 + 7T^3 + \ldots + (n^2 - (n-1)^2)T^{n-1}) - n^2 T^n \end{aligned}$$

hence

$$A = (I - T)^{-1}(I + 3T + 5T^2 + \ldots + (n^2 - (n-1)^2)T^{n-1}) - (I - T)^{-1} n^2 T^n$$

Writing

$$A = (I - T)^{-1} C - (I - T)^{-1} n^2 T^n$$

with

$$C = (I + 3T + 5T^2 + 7T^3 + \ldots + (n^2 - (n-1)^2)T^{n-1})$$

As

$$\begin{aligned} (I - T)C &= I + 3T + 5T^2 + 7T^3 + \ldots + (n^2 - (n-1)^2)T^{n-1}) \\ &\quad -T - 3T^2 - 5T^3 - 7T^4 - 9T^5 - \ldots \\ &\quad -((n-1)^2 - (n-2)^2)T^{n-1} - (n^2 - (n-1)^2)T^n \\ &= (I + 2T + 2T^2 + 2T^3 + 2T^4 \ldots + 2T^{n-1}) - (n^2 - (n-1)^2)T^n \end{aligned}$$

then

$$C = (I-T)^{-1}(I + 2T + 2T^2 + 2T^3 + \ldots + 2T^{n-1}) - (I-T)^{-1}(n^2 - (n-1)^2)T^n$$

Hence

$$\begin{aligned} A = (I-T)^{-1}((I-T)^{-1}(I + 2T + 2T^2 + 2T^3 + \ldots + 2T^{n-1}) \\ -(I-T)^{-1}(n^2 - (n-1)^2)T^n) - (I-T)^{-1}n^2T^n \end{aligned}$$

Expand the right hand side of A

$$\begin{aligned} A = (I-T)^{-1}((I-T)^{-1}(2I + 2T + 2T^2 + 2T^3 + \ldots + 2T^{n-1}) \\ +(I-T)^{-1}I - (I-T)^{-1}2I - (I-T)^{-1}(n^2 - (n-1)^2)T^n) \\ -(I-T)^{-1}n^2T^n \end{aligned}$$

then simplify

$$\begin{aligned} A = (I-T)^{-1}((I-T)^{-1}2(I-T)^{-1}(I-T^n) \\ +(I-T)^{-1}(I-2I) - (2n-1)(I-T)^{-1}T^n) \\ -n^2(I-T)^{-1}T^n. \end{aligned}$$

It follows that

$$\sum_{i=1}^{n} i^2T^{i-1} = 2(I-T)^{-3}(I-T^n) - (I-T)^{-2} - (2n-1)(I-T)^{-2}T^n - n^2(I-T)^{-1}T^n.$$

Hence using proposition 1

$$\lim_{n\to\infty} \sum_{i=1}^{n} i^2T^{i-1} = 2(I-T)^{-3} - (I-T)^{-2}.$$

The expectation $E(X^2)$ is thus formed:

$$E(X^2) = p_{(0,0)} + Q(1)\left(2(I-T)^{-3} - (I-T)^{-2} + 2(I-T)^{-2} + (I-T)^{-1}\right)R$$

$$= p_{(0,0)} + Q(1)\left(2(I-T)^{-3} + (I-T)^{-2} + (I-T)^{-1}\right)R.$$

The variance E_{var} of E_{rec} is therefore:

$$E_{var} = p_{(0,0)} + Q(1)\left(2(I-T)^{-3} + (I-T)^{-2} + (I-T)^{-1}\right)R - (E_{rec})^2.$$

4.4 Error Recovery Models Contrasted.

The Takishima *et al.* [13] model is a successor to the Maxted and Robinson model [10]. In the two models the n–step transition probabilities are defined differently: each model is defined for specific channel errors. In the Maxted and Robinson model, only one bit inversion error per codeword can occur, whereas in the Takishima *et al.* model, multiple bit inversion errors per codeword can occur.

The Takishima *et al.* model appears to be more realistic than that defined by Maxted and Robinson. The Takishima *et al.* model will therefore be used in this work.

Chapter 5

Computer Simulation.

5.1 Programming the Huffman Algorithm.

5.1.1 Data Structures Used.

The Huffman algorithm is applied to a sequence of n probabilities or frequencies, to store these values a $1 \times n$ array *Probability* is used:

	1	2	$\cdots$	n
Probability :	(p_i)	(p_j)	$\cdots$	(p_k)

with $Probability(i)$ denoting the ith entry in the array, so that $Probability(1) = (p_i)$ for example.

With n probabilities (or frequencies) there will be n codewords, to store each one a $1 \times n$ array *Code* is used:

	1	2	$\cdots$	n
Code :	(c_i)	(c_j)	$\cdots$	(c_k)

with entry $Code(i)$ denoting the string in the ith location.

Each codeword will have a specific length, to store each length a $1 \times n$ array *Length* is used. Let *Length*(i) denote the length of codeword *Code*(i) in location i:

	1	2	···	n
Length :	(l_i)	(l_j)	···	(l_k)

5.1.2 The General Algorithm.

The three arrays *Probability*, *Code* and *Length* are the main data structures used to program the Huffman algorithm. Using these data structures, the Huffman algorithm can be modelled to construct Huffman codes. The Huffman program is iterative; it can be explained in a number of steps. The example that follows considers probabilities, we could start with frequencies in a similar way:

1. The program user enters n probabilities in any order. Each probability p_m is labelled by a number and sequentially stored in the array *Probability*. The label is a unique number starting from 1 upwards, that is used to identify a particular probability. Initially all n probabilities are classified as unique and are therefore labelled from 1 to n:

	1	2	···	n
Probability:	$(p_i)_1$	$(p_j)_2$	···	$(p_k)_n$

   This completes the initial state of the *Probability* array.

2. After initialising the *Probability* array the *Code* and *Length* arrays are initialised. No codewords are generated at this point of the algorithm, therefore the entries in *Code* are defined to be empty:

	1	2	···	n
Code :	λ	λ	···	λ

   with λ representing the null string.

Consequently the length of each codeword in the *Length* array is set to zero:

	1	2	···	n
Length :	0	0	···	0

3. Following the initialisation of data structures, the program searches for the two smallest probabilities $(p_i)_x$ and $(p_j)_y$ in the *Probability* array. The two probabilities $(p_i)_x$ and $(p_j)_y$ will have different labels to identify them as being different, this implies that $x \neq y$. Let $(p_i)_x = (p_i)_1$ and $(p_j)_y = (p_j)_2$:

	1	2	···	n
Probability:	$(p_i)_1$	$(p_j)_2$	···	$(p_k)_n$

The smallest probability is deemed to be $(p_i)_1$, then let $(p_q)_{n+1}$ be the value $(p_q)_{n+1} = (p_i)_1 + (p_j)_2$ that is the sum of the two smallest probabilities labelled by $n+1$ that is the successor to n.

4. For each occurence of $(p_i)_1$ in the *Probability* array, replace each one by the probability $(p_q)_{n+1}$ and append bit 0 to the string in each respective location of *Code*:

	1	2	···	n
Probability :	$(p_q)_{n+1}$	$(p_j)_2$	···	$(p_k)_n$

	1	2	···	n
Code :	0	λ	···	λ

and for each occurence of $(p_j)_2$ in the *Probability* array replace it with the probability $(p_q)_{n+1}$ and append bit 1 to the string in each respective location of *Code*:

	1	2	···	n
Probability :	$(p_q)_{n+1}$	$(p_q)_{n+1}$	···	$(p_k)_n$

	1	2	···	n
Code :	0	1	···	λ

5. Update the *Length* array:

	1	2	···	n
Length :	1	1	···	0

6. At this point of the program the algorithm repeats, starting from step (3.): find the two smallest probabilities in the *Probability* array, both with different labels.

When the sum of the last two probabilities equal 1 the program will reverse all strings in the *Code* array to obtain an array of codewords. Each string in *Code* is equal to a codeword when the string is read from right to left, each string must therefore be reversed if the decoder decodes codewords from left to right. The program then terminates after reordering all codewords in increasing order of length.

5.1.3 Simulation Example.

1. The program user enters four probabilities: 0.1, 0.1, 0.3, 0.5, then each one is automatically labelled and stored in the 1×4 *Probability* array:

	1	2	3	4
Probability :	$(0.1)_1$	$(0.1)_2$	$(0.3)_3$	$(0.5)_4$

2. The arrays *Code* and *Length* are initialised:

	1	2	3	4
Code :	λ	λ	λ	λ

	1	2	3	4
Length :	0	0	0	0

3. In the *Probability* array search for the two smallest probabilities:

	1	2	3	4
Probability :	$\underline{(0.1)_1}$	$\underline{(0.1)_2}$	$(0.3)_3$	$(0.5)_4$

they are $(0.1)_1$ and $(0.1)_2$, as underlined.

4. Calculate the sum of probabilities $(0.1)_1$ and $(0.1)_2$ to produce the probability $(0.2)_5$ that is also labelled by the unique number '5'.

5. For each occurence of $(0.1)_1$ in the *Probability* array, replace it by the probability $(0.2)_5$ and append bit 0 to the string in each respective location of *Code*, as underlined here:

	1	2	3	4
Probability :	$(0.2)_5$	$(0.1)_2$	$(0.3)_3$	$(0.5)_4$
Code :	$\underline{0}$	λ	λ	λ

Similarly for each occurence of $(0.1)_2$ in the *Probability* array, replace it by the probability $(0.2)_5$ and append bit 1 to the string in each respective location of *Code*:

	1	2	3	4
Probability :	$(0.2)_5$	$(0.2)_5$	$(0.3)_3$	$(0.5)_4$
Code :	0	$\underline{1}$	λ	λ

6. Update the *Length* array:

	1	2	3	4
Length :	1	1	0	0

7. The algorithm repeats. In the *Probability* array, search for the two smallest probabilities with different labels:

	1	2	3	4
Probability :	$\underline{(0.2)_5}$	$\underline{(0.2)_5}$	$\underline{(0.3)_3}$	$(0.5)_4$

i.e. $(0.2)_5$ and $(0.3)_3$.

8. The sum of probabilities $(0.2)_5$ and $(0.3)_3$ is $(0.5)_6$, also labelled by '6', the successor of '5'.

9. For each occurence of $(0.2)_5$ in the *Probability* array, replace it by the probability $(0.5)_6$ and append bit 0 to the string in each respective location of *Code*:

	1	2	3	4
Probability :	$(0.5)_6$	$(0.5)_6$	$(0.3)_3$	$(0.5)_4$
Code :	$\underline{00}$	$\underline{10}$	λ	λ

and similarly for each occurence of $(0.3)_3$ in the *Probability* array, replace it by the probability $(0.5)_6$ and append bit 1 to the string in each respective location of *Code*:

	1	2	3	4
Probability :	$(0.5)_6$	$(0.5)_6$	$(0.5)_6$	$(0.5)_4$
Code :	00	10	$\underline{1}$	λ

10. Update the *Length* array:

	1	2	3	4
Length :	2	2	1	0

11. The algorithm repeats. In the *Probability* array, search for the two smallest probabilities with different labels:

	1	2	3	4
Probability :	$\underline{(0.5)_6}$	$\underline{(0.5)_6}$	$\underline{(0.5)_6}$	$\underline{(0.5)_4}$

i.e. $(0.5)_6$ and $(0.5)_4$.

12. The sum of probabilities $(0.5)_6$ and $(0.5)_4$ is $(1.0)_7$, also labelled by '7', the successor of '6'.

13. For each occurence of $(0.5)_6$ in the *Probability* array, replace it by the probability $(1.0)_7$ and append bit 0 to the string in each respective location of *Code*:

	1	2	3	4
Probability :	$(1.0)_7$	$(1.0)_7$	$(1.0)_7$	$(0.5)_4$
Code :	$\underline{000}$	$\underline{100}$	$\underline{10}$	λ

then similarly for each occurence of $(0.5)_4$ in the *Probability* array, replace it by the probability $(1.0)_7$ and append bit 1 to the string in each respective location of *Code*:

	1	2	3	4
Probability :	$(1.0)_7$	$(1.0)_7$	$(1.0)_7$	$(1.0)_7$
Code :	000	100	10	$\underline{1}$

14. Update the *Length* array:

	1	2	3	4
Length :	3	3	2	1

15. The last sum of probabilities is 1.0, this means that the program reverses all strings in the *Code* array:

	1	2	3	4
Code :	000	001	01	1

to produce an array of codewords.

Hence for each probability in *Probability*(i) there corresponds a codeword *Code*(i):

	1	2	3	4
Probability :	0.1	0.1	0.3	0.5
Code :	000	001	01	1

5.2 Constructing a Huffman Equivalent Code.

5.2.1 Data Structures Used.

A $1 \times n$ array *Code* is used to store strings, with *Code*(i) denoting the string in location i of *Code*. The initial state of *Code* is defined: *Code*(1) = 0, *Code*(2) = 1, and *Code*(i) = λ for $i = 3, \ldots, n$, with λ denoting the null string:

	1	2	3	...	n
Code :	0	1	λ	...	λ

To record the length of each string in $Code(i)$ the $1 \times n$ array $Length$ is used. Let $Length(i)$ denote the length of string in $Code(i)$, then the initial state of $Length$ is defined: $Length(1) = 1$, $Length(2) = 1$, and $Length(i) = 0$ for $i = 3, \ldots, n$:

	1	2	3	$\cdots$	n
$Length$:	1	1	0	$\cdots$	0

The $Code$ array will be used to store codewords, to know when a codeword has been generated during program execution it is necessary to assign to each codeword in $Code(i)$ a Boolean variable. The $1 \times n$ array $Valid$ assigns to each string $Code(i)$ the Boolean value $Valid(i)$: if $Code(i)$ is a codeword then $Valid(i)$=$True$, otherwise $Valid(i)$=$False$. The initial state of $Valid$ is defined: $Valid(i)$=$False$ for $i = 1, \ldots, n$:

	1	2	3	$\cdots$	n
$Valid$:	$False$	$False$	$False$	$\cdots$	$False$

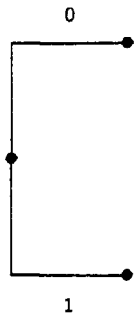
This completes the initialisation of arrays used in this program.

During program execution the array $Code$ will model the process of extending nodes, it is therefore necessary to define how this is achieved. Let the variable Max record the array location where $Code(Max + 1) = \lambda$. Thus effectively $Max + 1$ *points* to the location of an empty string in $Code$. Initially Max points to $Code(2)$:

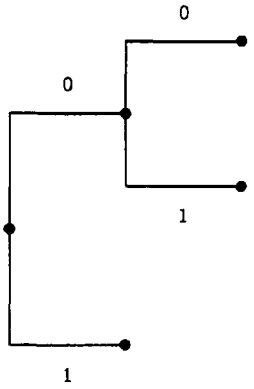
	1	2	3	$\cdots$	n
$Code$:	0	1	λ	$\cdots$	λ
		$\uparrow Max$			

When new strings are constructed, $Max + 1$ will identify an empty location until all entries in $Code$ are non-empty.

New strings are constructed following the operation of node extensions: if $\bar{s}$ is a string, then strings $\bar{s}0$ and $\bar{s}1$ are formed. For example, the two values $Code(1) = 0$ and $Code(2) = 1$ represents the simple binary tree that is the initial state of the program:



If a 0–node (defined in Section 2.2.1) was extended, to produce:



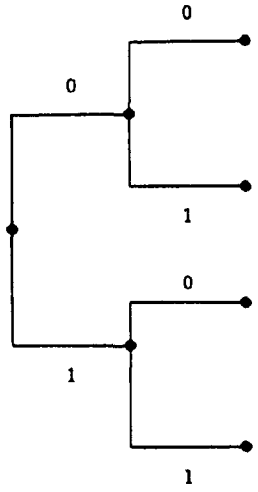
one more string is constructed, forming the set $\{01, 00, 1\}$. To represent this in the *Code* array, the 0–node $Code(1)$ is extended, that is $Code(1) = 00$. The next empty location $Code(3)$ is also written to, that is $Code(3) = 01$. Following the 0–node extension in the initial state, the *Code* array becomes:

	1	2	3	···	n
Code :	00	1	01	···	λ
			↑ Max		

Continuing, extending the 1–node in $Code(2)$ corresponds to $Code(2) = 10$ and $Code(4) = 11$:

	1	2	3	4	···	n
Code :	00	10	01	11	···	λ
				↑ Max		

that represent the 1–node extension in the binary tree:



5.2.2 The General Algorithm.

The general algorithm to construct Huffman equivalent codes is described here as a sequence of procedures. Initially the program user enters a length vector $(n_1, n_2, \ldots, n_M)$ and a synchronizing codeword 01_{r-1} or $01_{r-2}0$, then specifies which nodes to terminate. Following this, the program executes:

1. Initialise all data structures:

   (a) $Code(1) = 0$; $Code(2) = 1$, and for $i = 3, \ldots, n$: $Code(i) = \lambda$.

   (b) $Length(1) = 1$; $Length(2) = 1$, and for $i = 3, \ldots, n$: $Length(i) = 0$.

   (c) For $i = 1, \ldots, n$: $Valid(i)$=$False$.

   (d) Let $Max = 2$.

2. Extend all strings that are not codewords:

   Procedure *Extend Nodes* is

   UpperBound:=Max;

   for i from 1 ... UpperBound loop

   if Valid(i)=false then

   String:=$Code(i)$

   $Code(i)$:=$String \odot 0$;

   $Length(i) := Length(i) + 1$

   max:=max+1;

   $Code(Max)$:=$String \odot 1$;

   $Length(Max) := Length(i)$

   end if; end loop;

3. Search for c–nodes:

Procedure *Find c–nodes* is
for i from $1 \ldots Max$ loop
if $Valid(i)$=$False$ then
if $Code(i)$=c–node then
$Valid(i) := True$;
end if; end if; end loop;

4. Search for d–nodes and extend:

Procedure *Find d–nodes* is
for i from $1 \ldots Max$ loop
if $(Valid(i) = False)$ then
if $Code(i)$=d–node then
Extend $Code(i)$;
Max:=Max+1;
Length(i):=Length(i)+1;
Length(Max):=Length(i);
end if; end if;
end loop;

5. Read the length vector $(n_1, n_2, \ldots, n_M)$. If n codewords of length i are required the program searches for the number of c–nodes of length i found so far:

for j in 1...Max loop
counter:=0;
if $(Valid(j) = True$ and $Length(j) = i)$ then
counter:=counter+1; end if;
end loop;

If more nodes need terminating, the program initially searches for 0–nodes or 1–nodes, depending on the user's initial choice. If 0–nodes are terminated initially, then apply the procedure:

for i in 1...Max loop

Acounter:=counter;

if $(Valid(i) = False)$ and $(Code(i)$=0–node) then

Acounter:=Acounter+1;

end if;

end loop;

until *Acounter* is equal to the number of codewords n of length i.

Otherwise, to terminate 1–nodes apply the procedure:

for i in 1...Max loop

Acounter:=counter;

if $(Valid(i) = False)$ and $(Code(i)$=1–node) then

Acounter:=Acounter+1;

end if;

end loop;

until *Acounter* is equal to the number of codewords n of length i.

If there is an insufficient number of 0–nodes terminated, the program automatically searches for 1–nodes, and vice versa. When there is an insufficient number of nodes remaining, the program terminates and returns the message that the method has failed.

6. After terminating the correct number of nodes, the program repeats execution from step (3.), then terminates when a Huffman equivalent code is constructed or until the method fails.

5.2.3 Simulation Example.

A Huffman equivalent code is to be constructed. The program user enters the length vector (0,1,4,4) and a synchronizing codeword 011. Let the program terminate 0–nodes when possible:

1. Initialise all data structures:

	1	2	3	4	5	6	7	8	9
Code :	0	1	λ	λ	λ	λ	λ	λ	λ
		$\uparrow Max$							

Length :	1	1	0	0	0	0	0	0	0

Valid :	*False*	*False*	*False*	*False*	*False*	*False*	*False*	*False*	*False*

2. In *Code*, extend all strings that are not codewords. That is: string 0 in *Code*(1) is replaced by 00, and λ in *Code*(3) is replaced by 01:

	1	2	3	4	5	6	7	8	9
Code :	00	1	01	λ	λ	λ	λ	λ	λ
			↑ *Max*						

then string 1 in *Code*(2) is replaced by 10, and λ in *Code*(4) is replaced by 11:

	1	2	3	4	5	6	7	8	9
Code :	00	10	01	11	λ	λ	λ	λ	λ
				↑ *Max*					

Update the *Length* array:

	1	2	3	4	5	6	7	8	9
Length :	2	2	2	2	λ	λ	λ	λ	λ

3. Search for c–nodes. A c–node is located in *Code*(4). To ensure that codeword *Code*(4) is not altered during the program run, let *Valid*(4)=*True*. Hence one codeword of length 2 is found:

	1	2	3	4	5	6	7	8	9
Code :	00	10	01	11	λ	λ	λ	λ	λ
				↑ *Max*					

$Valid$:	$False$	$False$	$False$	$\underline{True}$	$False$	$False$	$False$	$False$	$False$

4. Extend all d–nodes in $Code$. One d–node is present: the string $Code(3)$, it is therefore extended, and $Length(3)$ updated:

	1	2	3	4	5	6	7	8	9
$Code$:	00	10	010	11	011	λ	λ	λ	λ
					$\uparrow Max$				
$Length$:	2	2	3	2	3	λ	λ	λ	λ

It is specified in the length vector (0,1,4,4) that one codeword of length 2 is required. One codeword $Code(4)$ of length 2 has already been found, therefore no more are required at this stage. To continue, extend all remaining 1–nodes and 0–nodes:

	1	2	3	4	5	6	7	8	9
$Code$:	$\underline{000}$	$\underline{100}$	010	11	011	$\underline{001}$	$\underline{101}$	λ	λ
							$\uparrow Max$		

5. Search for c–nodes. A c–node is located in $Code(5)$. To ensure that codeword $Code(5)$ is not altered during the program run, let $Valid(5)$=$True$. Hence one codeword of length 3 is found:

	1	2	3	4	5	6	7	8	9
$Valid$:	$False$	$False$	$False$	$True$	$\underline{True}$	$False$	$False$	$False$	$False$

6. Extend all d–nodes in $Code$. Two d–nodes are present, extend the strings $Code(6)$ and $Code(7)$:

	1	2	3	4	5	6	7	8	9
$Code$:	000	100	010	11	011	$\underline{0010}$	$\underline{1010}$	$\underline{0011}$	$\underline{1011}$
									$\uparrow Max$

then update the *Length* array:

	1	2	3	4	5	6	7	8	9
Length :	3	3	3	2	3	4	4	4	4

The length vector (0,1,4,4) specifies that four codewords of length 3 are required. One codeword *Code*(5) of length 3 has already been found. To find three codewords of length 3, the program searches sequentially from 1 to *Max* in the *Code* array until 0–nodes are found. The three codewords found are: *Code*(1), *Code*(2), and *Code*(3). The *Valid* array is then updated:

	1	2	3	4	5	6	7	8	9
Valid :	*True*	*True*	*True*	*True*	*True*	*False*	*False*	*False*	*False*

To continue, extend all remaining 1–nodes and 0–nodes. There are no 0–nodes or 1–nodes to extend at this stage.

7. Search for *c*–nodes. Two *c*–nodes are located: *Code*(8) and *Code*(9). To ensure that codewords *Code*(8) and *Code*(9) are not altered during the program run, let *Valid*(8)=*True* and *Valid*(9)=*True*. Hence two codewords of length 4 are found:

	1	2	3	4	5	6	7	8	9
Valid :	*True*	*True*	*True*	*True*	*True*	*False*	*False*	*True*	*True*

8. Extend all *d*–nodes in *Code*. No *d*–nodes are found, therefore continue. The length vector (0,1,4,4) specifies that four codewords of length 4 are required. Two codewords *Code*(8) and *Code*(9) of length 4 are known. To find two more codewords of length 4, the program searches sequentially from 1 to *Max* for 0–nodes. Two 0–nodes, *Code*(6) and *Code*(7), are found:

	1	2	3	4	5	6	7	8	9
Code :	000	100	010	11	011	0010	1010	0011	1011
									↑ *Max*

then the *Valid* array is updated:

	1	2	3	4	5	6	7	8	9
Valid :	*True*	*True*	*True*	*True*	*True*	*True*	*True*	*True*	*True*

The program observes that for $i = 1, \ldots, n$, $Valid(i)$=$True$, signifying that all codewords are generated. The program terminates after constructing an array of codewords:

	1	2	3	4	5	6	7	8	9
Code :	000	100	010	11	011	0010	1010	0011	1011

and ordering the array contents in increasing order of length, in preparation for the user:

	1	2	3	4	5	6	7	8	9
Code :	11	011*	100	010	000	0011*	1011*	0010	1010

This completes the construction of a Huffman equivalent code. To assign probabilities to each codeword an array of probabilities $Prob$ is constructed. The program user will specify that codeword $Code(i)$ is assigned the probability p_i in $Prob$, by the assignment $Prob(i) = p_i$:

	1	2	3	4	5	6	7	8	9
Code :	11	011*	100	010	000	0011*	1011*	0010	1010
Prob :	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9

5.3 Constructing T–Codes.

To construct a T–code two different programs are required: one to construct simple T–codes and the other to construct generalised T–codes. Both programs process input files in order to construct

a file containing codewords from a T–code. Combinations of simple and generalised augmentations can be performed.

5.3.1 Programming the Simple T–Code Algorithm.

Two files *Code* and *Double* are used to construct a simple T–Code. Both files are used in the program as follows:

1. Initialise the two files. The initial state of *Code* is defined to contain two strings, 0 and 1. The file *Double* is empty.
2. The program user enters an augmentation level n.
3. Display the contents of *Code* and select a word $\bar{x}$ to be the next prefix:

Code
0
1

4. Overwrite the *Double* file with the contents of *Code*; writing the contents twice:

Double
0
1
0
1

5. In the first half of *Double*, remove the string $\bar{x}$ and append it to the left of all strings in the second half of *Double*. For example, if $\bar{x}$=0:

Double
1
00
01

6. Rename the *Double* file to *Code*. This completes the first augmentation, the new *Code* file is a T–Code with three codewords:

Code
1
00
01

To form further augmentations, repeat the procedure from step (3.) with the new *Code* file until the nth augmentation is attained.

The program terminates after reordering all codewords in *Code* in increasing order of length.

5.3.2 Simulation Example.

Construct a simple T–Code from two augmentations:

1. The initial state of *Code* is defined:

   Code

   0

   1

   with *Double* empty:

   Double

2. The program user enters $n = 2$ augmentations.

3. From *Code* select 1 to be a prefix word:

   Code

   0

   $\underline{1}$

4. Overwrite the *Double* file with the contents of *Code*; writing the contents twice:

   Double

   0

   1

   0

   1

5. In the first half of *Double*, remove the prefix word 1 and append it to the left of all strings in the second half of *Double*:

   Double

   0

   10

   11

6. Rename the *Double* file to *Code*. This completes the first augmentation.

Code

0

10

11

7. The second augmentation is applied. From *Code* select 10 to be a prefix word:

Code

0

<u>10</u>

11

8. Overwrite the *Double* file with the contents of *Code*; writing the contents twice:

Double

0

10

11

0

10

11

9. In the first half of *Double*, remove the prefix word 10 and append it to the left of all strings in the second half of *Double*:

Double

0

11

100

1010

1011

10. Rename the *Double* file to *Code*. This completes the second augmentation, a T-Code is constructed:

Code

0

11

100

1010

1011

Following the T–code construction the user has the option to apply node reduction/extension operations. To apply node reduction the user selects a codeword c of length n from *Code*, the program searches for two codewords c and c_1 with a prefix r of length $n-1$ that is equal to the prefix of c. If this search algorithm is successful both c and c_1 are removed from *Code* and r is written to *Code*, otherwise no action is taken. Node extension requires the user to select a codeword c. c is removed from *Code*, followed by the inclusion of codewords $c0$ and $c1$.

5.3.3 Programming the Generalised T–Code Algorithm.

Three files are used to construct a generalised T–code. The *Code* file initially contains a column of strings $\bar{x}, \bar{y}, \ldots, \bar{z}$ that are codewords:

Code
$\bar{x}$
$\bar{y}$
$\vdots$
$\bar{z}$

A temporary storage file *Temp* is required to store strings during program execution. Initially *Temp* is empty. Similarly the third file *Storage* is initially empty, and will be used to collect codewords during the program run.

To construct a generalised T–code with the files *Code*, *Temp*, and *Storage*, apply the following algorithm:

1.
   - Let i denote a counter, with $i = 0$ initially.
   - The program user selects a prefix word by choosing a string $\bar{w}$ from *Code*.
   - The user enters a number k for the total number of iterations.
   - For all strings in *Code*, except the selected prefix word, write each one to *Temp* and *Storage*:

Temp	*Storage*
$\bar{x}$	$\bar{x}$
$\bar{y}$	$\bar{y}$
$\vdots$	$\vdots$
$\bar{z}$	$\bar{z}$

2. Increment i by 1: $i := i + 1$. Append $\bar{w}$ to all strings in *Temp*:

$$\begin{array}{l} Temp \\ \bar{w}\bar{x} \\ \bar{w}\bar{y} \\ \vdots \\ \bar{w}\bar{z} \end{array}$$

then add the contents of *Temp* to *Storage*:

$$\begin{array}{l} Storage \\ \bar{x} \\ \bar{y} \\ \vdots \\ \bar{z} \\ \bar{w}\bar{x} \\ \bar{w}\bar{y} \\ \vdots \\ \bar{w}\bar{z} \end{array}$$

3. • If $i < k$ then repeat from step (2.):

Increment i by 1: $i := i + 1$. Append all strings in *Temp* with the prefix word $\bar{w}$:

$$\begin{array}{l} Temp \\ \bar{w}\bar{w}\bar{x} \\ \bar{w}\bar{w}\bar{y} \\ \vdots \\ \bar{w}\bar{w}\bar{z} \end{array}$$

then add the contents of *Temp* to *Storage*:

$$\begin{array}{l} \textit{Storage} \\ \bar{x} \\ \bar{y} \\ \vdots \\ \bar{z} \\ \bar{w}\bar{x} \\ \bar{w}\bar{y} \\ \vdots \\ \bar{w}\bar{z} \\ \bar{w}\bar{w}\bar{x} \\ \bar{w}\bar{w}\bar{y} \\ \vdots \\ \bar{w}\bar{w}\bar{z} \end{array}$$

- otherwise if $i = k$:

  Append $\bar{w}$ to all strings in *Temp*:

$$\begin{array}{l} \textit{Temp} \\ \bar{w}\bar{w}\bar{x} \\ \bar{w}\bar{w}\bar{y} \\ \vdots \\ \bar{w}\bar{w}\bar{z} \end{array}$$

  add the contents of *Temp* to *Storage*:

$$\begin{array}{l} \textit{Storage} \\ \bar{x} \\ \bar{y} \\ \vdots \\ \bar{z} \\ \bar{w}\bar{x} \\ \bar{w}\bar{y} \\ \vdots \\ \bar{w}\bar{z} \\ \bar{w}\bar{w}\bar{x} \\ \bar{w}\bar{w}\bar{y} \\ \vdots \\ \bar{w}\bar{w}\bar{z} \\ \bar{w}\bar{w}\ldots\bar{w} \end{array}$$

including the word $\bar{w}\bar{w}\ldots\bar{w}$, that is the concatenation of $\bar{w}$ $k+1$ times.

4. The program terminates after renaming *Storage* to *Code*. The *Code* file may then be used in the simple T–code program or again in this one.

5.3.4 Simulation Example.

The *Code* file:

Code
0
11
10

will be used to construct a T–code by applying the generalised T–code program:

1.
   - Let $i = 0$.
   - The program user selects the prefix word 0 from *Code*.
   - The number of iterations $k = 3$ is entered.
   - For all strings in *Code*, except the selected prefix word, write each one to *Temp* and *Storage*:

Code	*Temp*	*Storage*
0	11	11
11	10	10
10		

2. Increment i by 1: $i = 1$. Append 0 to all strings in *Temp*:

Temp
011
010

   then add the contents of *Temp* to *Storage*:

Storage
11
10
011
010

3. Increment i by 1: $i = 2$. Append 0 to all strings in *Temp*:

Temp
0011
0010

then add the contents of *Temp* to *Storage*:

Storage
11
10
011
010
0011
0010

4. Increment i by 1: $i = 3$. Append 0 to all strings in *Temp*:

Temp
00011
00010

add the contents of *Temp* to *Storage*:

Storage
11
10
011
010
0011
0010
00011
00010
0000

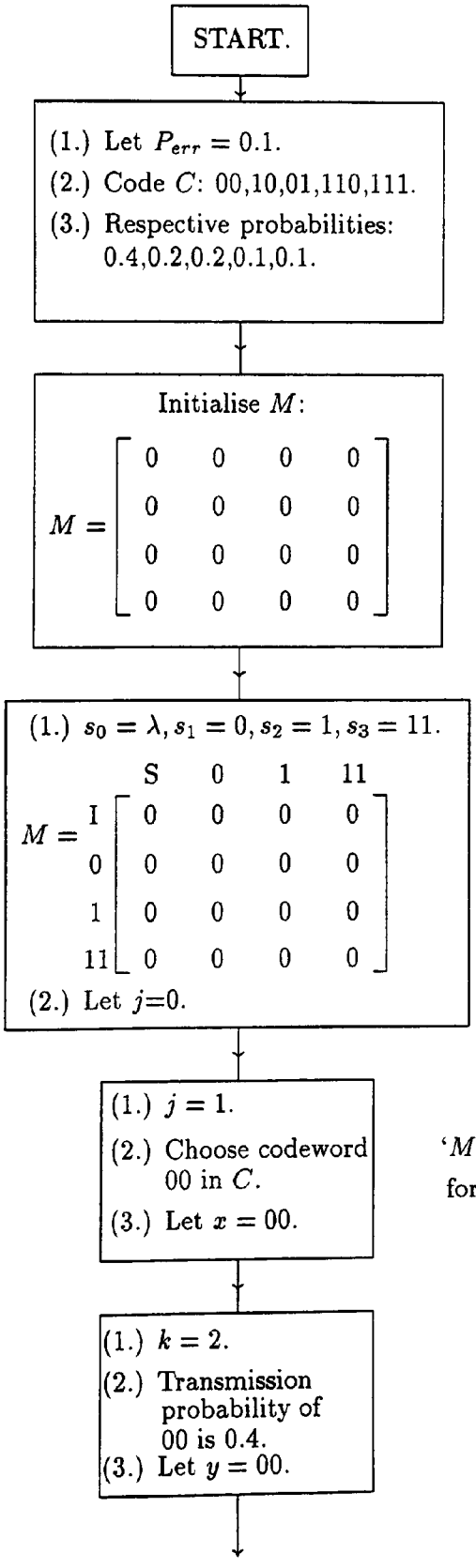
including the codeword 0000 that is the concatenation of the prefix word '0' four times.

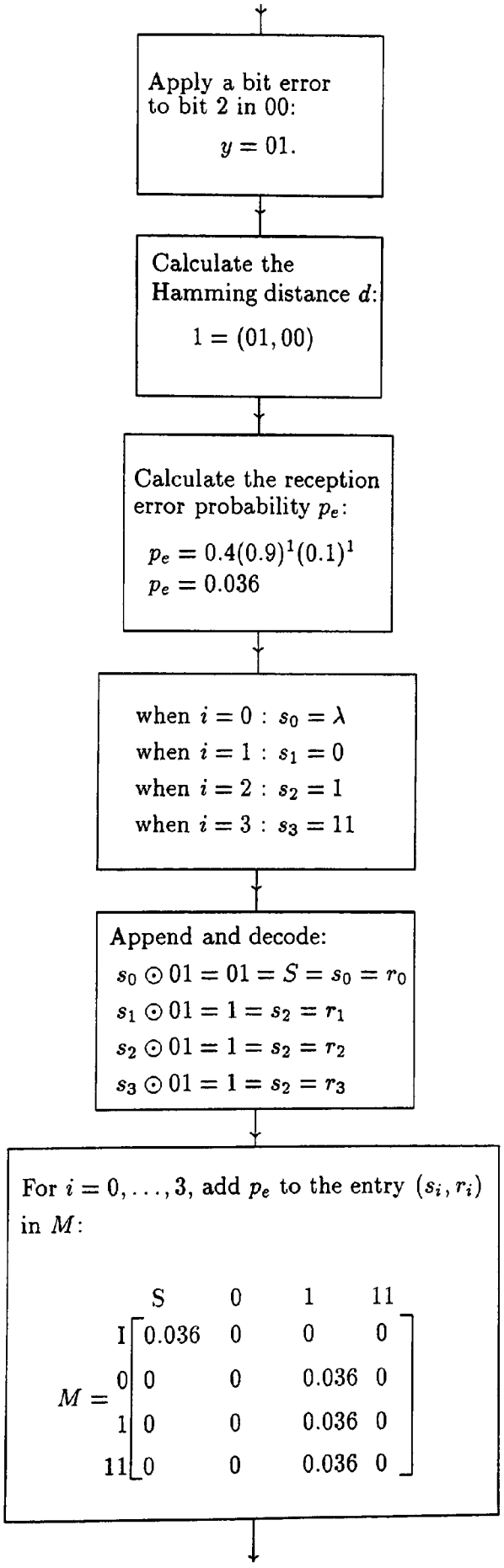
5. The program terminates after renaming *Storage* to *Code*.

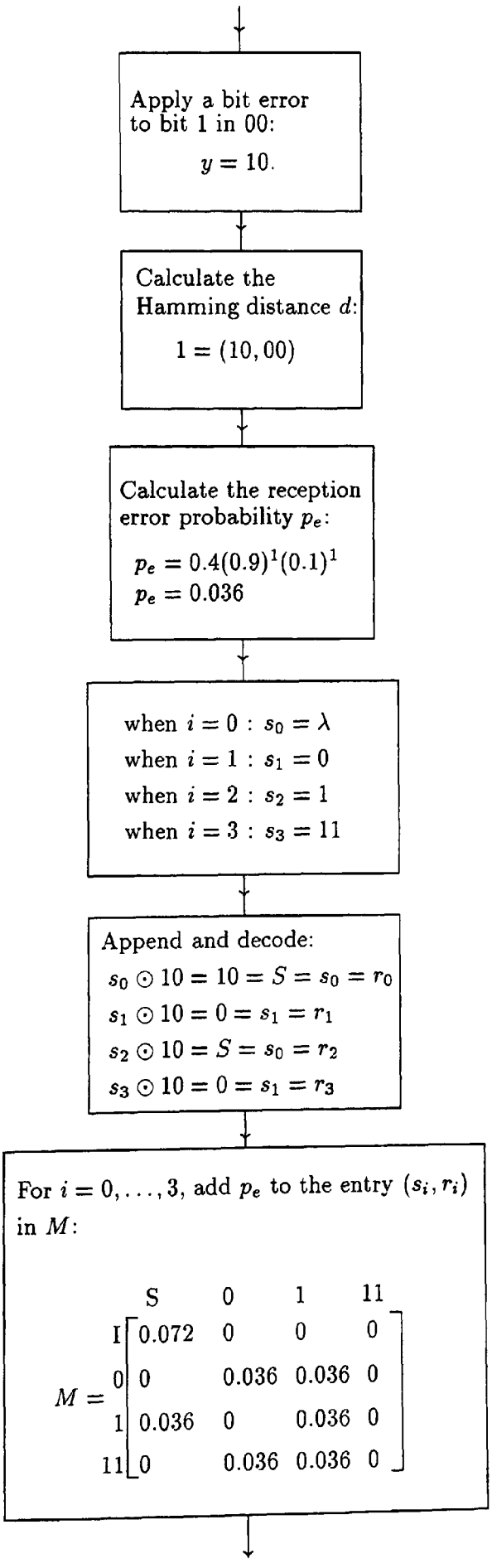
The *Code* file contains a generalised T–code. In practice the *Code* file can be used again to construct another generalised T–code.

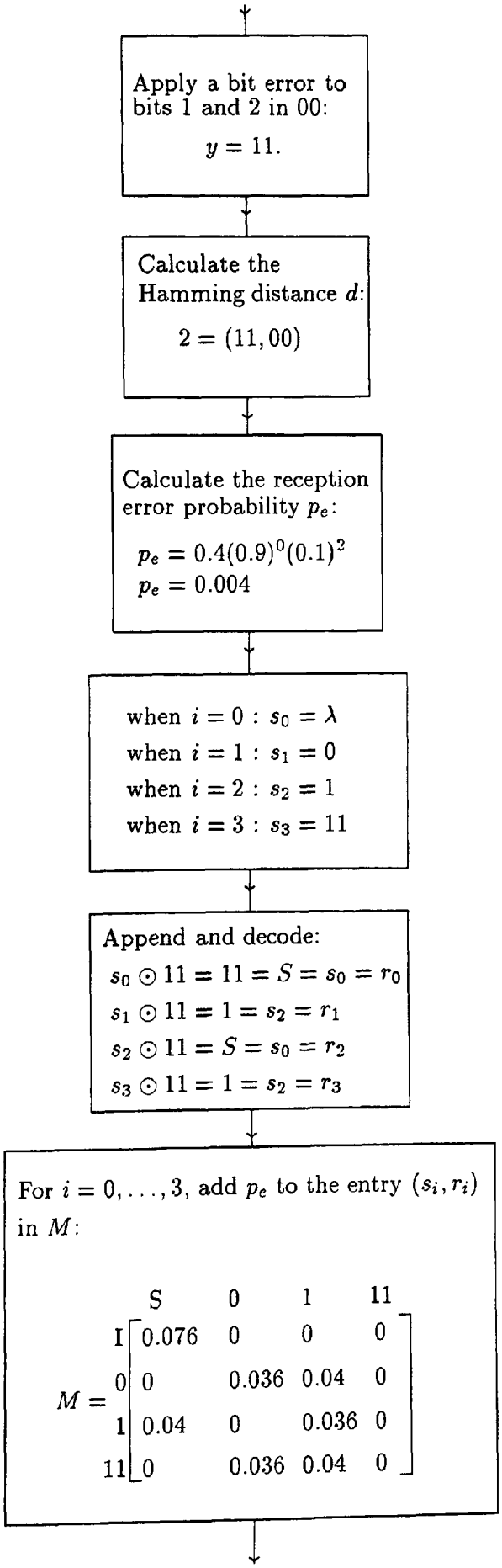
5.4 Algorithm to Generate a Transition Matrix.

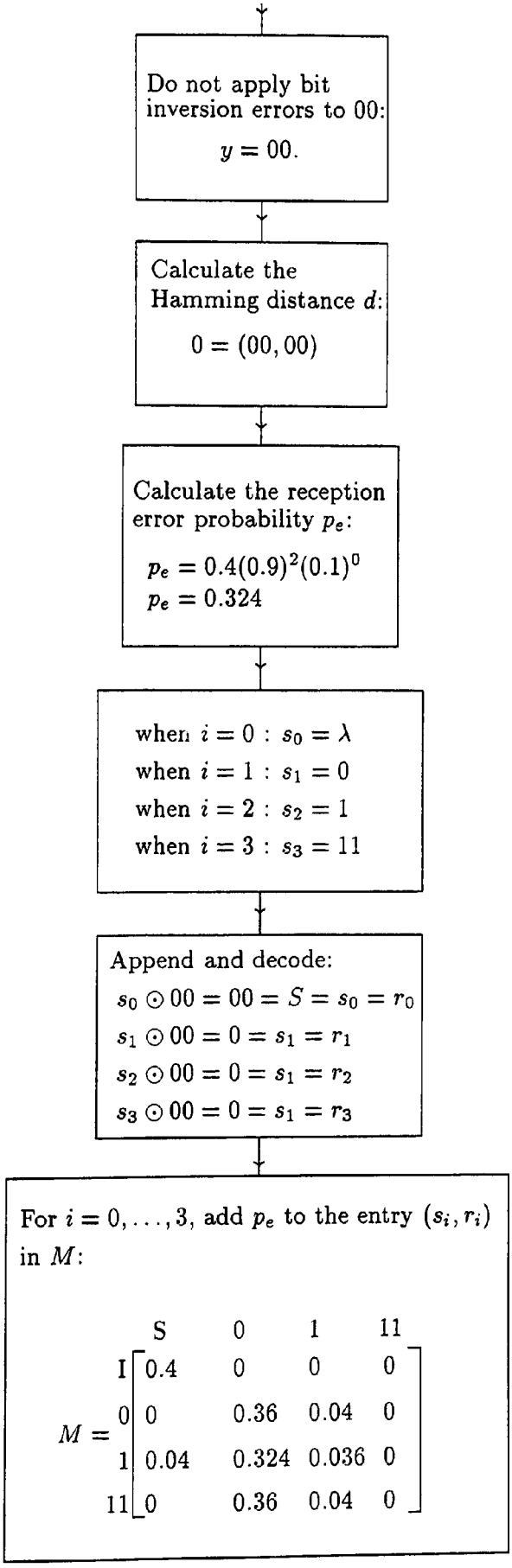
The algorithm presented to construct the transition matrix M, defined in Section 4.3, is explained here by an example. In this example one codeword is used to show the process that each codeword in a variable length code C is subject to.




Apply a bit error to bit 2 in 00:
$y = 01.$
Calculate the Hamming distance d:
$1 = (01, 00)$
Calculate the reception error probability p_e:
$p_e = 0.4(0.9)^1(0.1)^1$
$p_e = 0.036$
when $i = 0 : s_0 = \lambda$
when $i = 1 : s_1 = 0$
when $i = 2 : s_2 = 1$
when $i = 3 : s_3 = 11$
Append and decode:
$s_0 \odot 01 = 01 = S = s_0 = r_0$
$s_1 \odot 01 = 1 = s_2 = r_1$
$s_2 \odot 01 = 1 = s_2 = r_2$
$s_3 \odot 01 = 1 = s_2 = r_3$
For $i = 0, \ldots, 3$, add p_e to the entry (s_i, r_i) in M:
$M =$
S 0 1 11
I 0.036 0 0 0
0 0 0 0.036 0
1 0 0 0.036 0
11 0 0 0.036 0


Apply a bit error to bit 1 in 00:
$y = 10.$
Calculate the Hamming distance d:
$1 = (10, 00)$
Calculate the reception error probability p_e:
$p_e = 0.4(0.9)^1(0.1)^1$
$p_e = 0.036$
when $i = 0 : s_0 = \lambda$
when $i = 1 : s_1 = 0$
when $i = 2 : s_2 = 1$
when $i = 3 : s_3 = 11$
Append and decode:
$s_0 \odot 10 = 10 = S = s_0 = r_0$
$s_1 \odot 10 = 0 = s_1 = r_1$
$s_2 \odot 10 = S = s_0 = r_2$
$s_3 \odot 10 = 0 = s_1 = r_3$
For $i = 0, \ldots, 3$, add p_e to the entry (s_i, r_i) in M:
$M =$
S 0 1 11
I 0.072 0 0 0
0 0 0.036 0.036 0
1 0.036 0 0.036 0
11 0 0.036 0.036 0


Apply a bit error to bits 1 and 2 in 00:
$y = 11.$
Calculate the Hamming distance d:
$2 = (11, 00)$
Calculate the reception error probability p_e:
$p_e = 0.4(0.9)^0(0.1)^2$
$p_e = 0.004$
when $i = 0 : s_0 = \lambda$
when $i = 1 : s_1 = 0$
when $i = 2 : s_2 = 1$
when $i = 3 : s_3 = 11$
Append and decode:
$s_0 \odot 11 = 11 = S = s_0 = r_0$
$s_1 \odot 11 = 1 = s_2 = r_1$
$s_2 \odot 11 = S = s_0 = r_2$
$s_3 \odot 11 = 1 = s_2 = r_3$
For $i = 0, \ldots, 3$, add p_e to the entry (s_i, r_i) in M:
$M =$
S 0 1 11
I 0.076 0 0 0
0 0 0.036 0.04 0
1 0.04 0 0.036 0
11 0 0.036 0.04 0


Do not apply bit inversion errors to 00:
$y = 00$.
Calculate the Hamming distance d:
$0 = (00, 00)$
Calculate the reception error probability p_e:
$p_e = 0.4(0.9)^2(0.1)^0$
$p_e = 0.324$
when $i = 0$: $s_0 = \lambda$
when $i = 1$: $s_1 = 0$
when $i = 2$: $s_2 = 1$
when $i = 3$: $s_3 = 11$
Append and decode:
$s_0 \odot 00 = 00 = S = s_0 = r_0$
$s_1 \odot 00 = 0 = s_1 = r_1$
$s_2 \odot 00 = 0 = s_1 = r_2$
$s_3 \odot 00 = 0 = s_1 = r_3$
For $i = 0, \ldots, 3$, add p_e to the entry (s_i, r_i) in M:
$M =$
S 0 1 11
I 0.4 0 0 0
0 0 0.36 0.04 0
1 0.04 0.324 0.036 0
11 0 0.36 0.04 0

The program continues to construct the transition matrix M by branching back to the step labelled *Main*. Execution continues with (1.) j=2, (2.) $c_2 = 01$, (3.) $x = 01$. The program terminates after using the last codeword $c_5 = 111$.

5.5 Determining the Observed Average Error Recovery.

The error recovery value E_{rec} defined in Section 4.3 is an estimate of the synchronization delay, used to judge the robustness of a variable length code. Prior to using the statistic E_{rec} for such purposes, we must observe how accurate E_{rec} is in practice. This is possible by computer modelling.

The computer model simulates the procedure of encoding/decoding data and transmitting it over a noisy channel. This involves creating a text file F_1 to store information, then compressing it to form a data file. To simulate the affect of channel errors, the data file is subject to random bit inversions, and the resulting data is then decoded to a text file F_2. Following this procedure, an observed error recovery value is obtained by comparing the two files F_1 and F_2. The observed error recovery value is an average quantity that is compared to the theoretical value E_{rec}. To be useful E_{rec} should be a good approximation.

5.5.1 Error Recovery Example.

In this example the method of calculating an observed error recovery value is illustrated. Let $C = \{00, 01, 10, 110, 111\}$ be a variable length code used to encode a text file. Here the corresponding coded text file is termed a *code file*:

code file: | 00 | 01 | 10 | 10 | 110 | 110 | 10 | 111 | 01 | 00 | 110 | 110 | 111 | 10 | 01 |

with each codeword separated by '|' for clarity.

The code file is subject to random bit inversion errors. The designated bits to be inverted are underlined:

code file: | 00 | 01 | $\underline{1}$0 | 10 | 110 | 110 | 1$\underline{0}$ | 111 | $\underline{0}$1 | 00 | 110 | 110 | 111 | 1$\underline{0}$ | 01 |

The resulting file is termed an *error file*. Thus the error file in this example is defined:

error file: | 00 | 01 | 00 | 10 | 110 | 110 | 111 | 111 | 10 | 01 | 10 | 110 | 111 | 110 | 1

Both *code* and *error* files are used to calculate the observed error recovery value, as detailed in the following algorithm:

1. Start to decode both files, one codeword at a time, until the first error is located. One error is located, as underlined:

   The code file: | 00 | 01 | 10 |

   The error file: | 00 | 01 | $\underline{0}$0 |

   At this point, synchronization is lost.

2. The two codewords 10 and 00 from both files are equal in length, in this case synchronization is regained in one codeword, this is the codeword 10 in *code* file, as overlined:

   The code file: | 00 | 01| $\overline{10 |}$

   The error file: | 00 | 01| $\underline{0}$0 |

3. Repeat the algorithm. Continue to decode files until an error is located:

   The code file: | 00 | 01| $\overline{10 |}$ 10 | 110 | 110 | 10 |

   The error file: | 00 | 01| $\underline{0}$0 | 10 | 110 | 110 | 1$\underline{1}$1 |

   At this point the two codewords 10 and 111 differ in length, this implies that synchronization will require more than one codeword.

4. Decode files until synchronization is established, or when the code file has come to an end:

   The code file: | 00 | 01| $\overline{10 |}$ 10 | 110 | 110 | 10 | 111 | 01 | 00 | 110 |

   The error file: | 00 | 01| $\underline{0}$0 | 10 | 110 | 110 | 1$\underline{1}$1 | 11$\underline{1}$ | 10 | 01 | 10 |

   Hence synchronization is established in five codewords in the *code* file:

   The code file: | 00 | 01| $\overline{10 |}$ 10 | 110 | 110 $\overline{| 10 | 111 | 01 | 00 | 110 |}$

   The error file: | 00 | 01| $\underline{0}$0 | 10 | 110 | 110 | 1$\underline{1}$1 | 11$\underline{1}$ | 10 | 01 | 10 |

5. Repeat the algorithm. Decode files until an error is located:

The code file: | 00 | 01| 10 | 10 | 110 | 110 | 10 | 111 | 01 | 00 | 110 | 110 | 111 | 10 |

The error file: | 00 | 01| 00 | 10 | 110 | 110 | 111 | 111 | 10 | 01 | 10 | 110 | 111 | 110 |

At this point synchronization is lost.

6. Decode files until synchronized:

The code file: | 00 | 01| 10 | 10 | 110 | 110 | 10 | 111 | 01 | 00 | 110 | 110 | 111 | 10 | 01 |

The error file: | 00 | 01| 00 | 10 | 110 | 110 | 111 | 111 | 10 | 01 | 10 | 110 | 111 | 110 | 1

Synchronization is improperly established due to the end of the *code* file, in this case we interpret the number of codewords read by the decoder, from the start of losing synchronization to the end of the *code* file, to be a true value. The number of codewords to resynchronize is then 2:

The code file: | 00 | 01| 10 | 10 | 110 | 110 | 10 | 111 | 01 | 00 | 110 | 110 | 111 | 10 | 01 |

The error file: | 00 | 01| 00 | 10 | 110 | 110 | 111 | 111 | 10 | 01 | 10 | 110 | 111 | 110 | 1

7. After decoding the *code* file the observed average error recovery value is calculated. Due to the effect of four errors, synchronization was lost three times, in each case the number of codewords taken to resynchronize the code was: 1, 5, and 2. Hence an observed error recovery value is calculated:

$$\frac{1+5+2}{3} = 2.67$$

from the formula $\bar{x}$:

$$\bar{x} = \frac{x_1 + x_2 + \ldots + x_n}{n}$$

where x_i denotes the number of codewords required to regain synchronization when it becomes lost, and n the total number of times synchronization is lost.

Additionally, the variance s^2 can be calculated from the formula:

$$s^2 = \frac{1}{n}\left[\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right]$$

i.e.

$$s^2 = \frac{1}{3}[30 - 3(7.11)] = 2.89$$

Chapter 6

Simulation Results.

6.1 Example of Interpreting Robustness.

The following example will illustrate the method of collecting simulation data. The procedure of the simulation model involves compressing a text sample (from [2]) with a variable length code, and applying random errors to the compressed text, to obtain an observed error recovery value. This procedure is iterated 100 times to generate 100 observed values.

For this example the text sample used is the novel *Jane Eyre* by Charlotte Bronte. Initially the simulation model is applied with a Huffman code. The specific code will be derived from the text statistics of the sample. This involves determining the frequency of each source symbol in the text to apply the Huffman algorithm. However not all source symbols in the sample are used. Symbols not equal to a letter in the English alphabet, or the '*space*' symbol, are removed.

Following the removal of unwanted source symbols, the frequency of each symbol is used to construct a Huffman code. Table 6.1 contains the Huffman code C_1 that was derived from the frequencies of source symbols.

The simulation model is applied: the text sample *Jane Eyre* is compressed and random errors are inserted (with $P_{err} = 0.0001$), to obtain 100 observed error recovery values. The graph on page 129 represents these observations. The observed error recovery average is 3.697. Using the code C_1 and transmission probabilities in Table 6.1 the theoretical error recovery is calculated using the formula E_{rec} defined in pages 76 to 77: $E_{rec} = 3.719$. Comparing the observed error recovery average with E_{rec}, the theoretical value is a good estimate in this case. Considering the variance, the average

Symbol	Transmission Probability	C_1
(*space*)	0.190617	111
e	0.103395	010
t	0.0688609	1011
a	0.0645354	1010
o	0.0622213	1001
i	0.0575601	1000
n	0.055747	0110
s	0.0514721	0011
r	0.0486622	0010
h	0.0471198	0001
d	0.0384853	11010
l	0.033386	11000
u	0.0241265	01110
m	0.0228879	00000
c	0.0192597	110111
w	0.0192339	110110
y	0.0177917	110010
f	0.0173309	011111
g	0.0155024	011110
p	0.0124589	000011
b	0.0114496	000010
v	0.00782971	1100110
k	0.00618608	11001111
x	0.00131408	110011100
j	0.00125519	1100111010
q	0.000974194	11001110111
z	0.000337817	11001110110

Table 6.1: Text statistics used to construct C_1.

of the observed variance is 12.524. The theoretical variance of E_{rec} is 13.043. Comparing observed and theoretical variances, the theoretical variance is a reasonable estimate in this case.

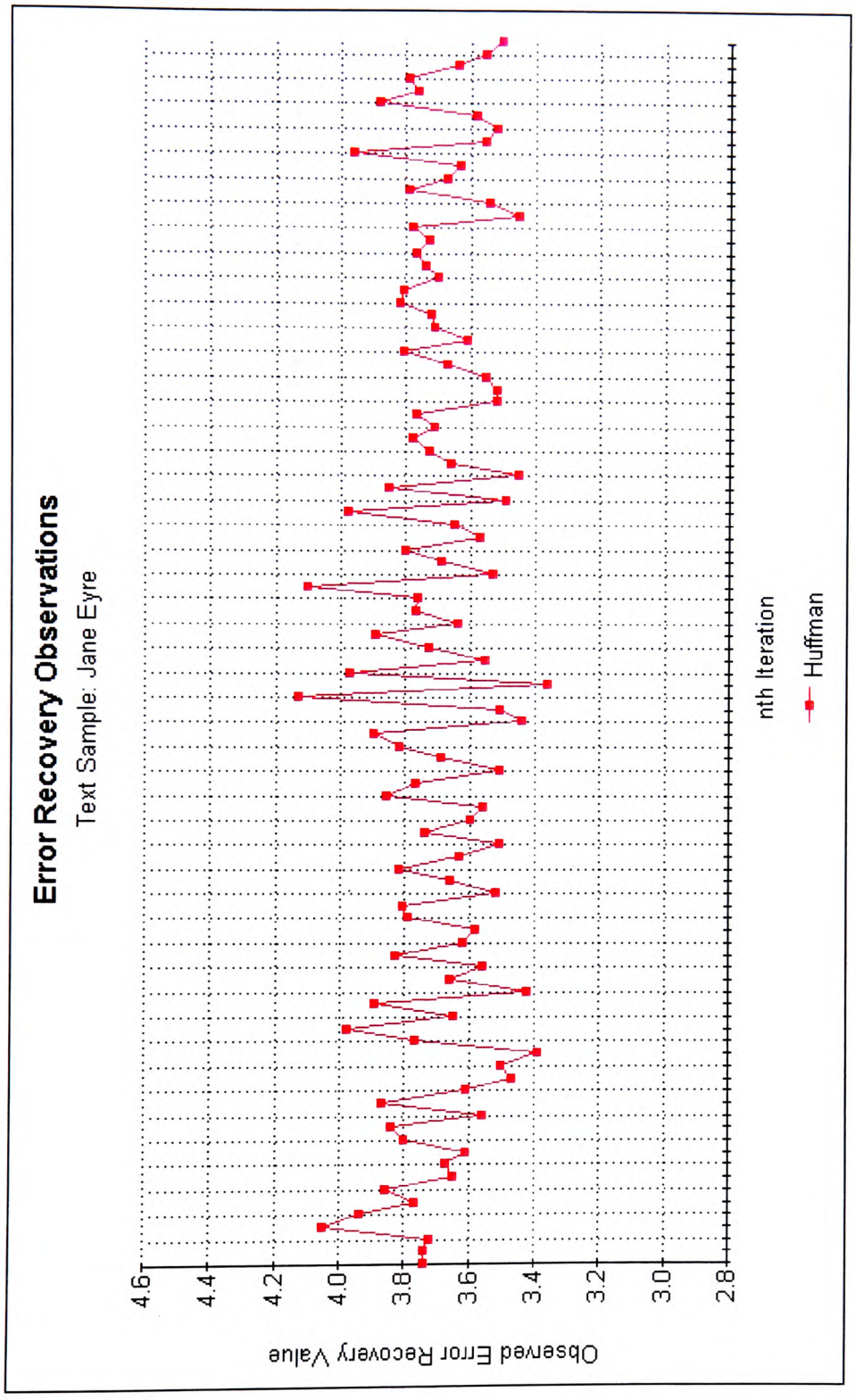


Figure 6.1: Observation results example.

Interpreting the robustness of C_1 requires comparison with other simulation results from other codes. A Huffman equivalent code is used to obtain theoretical and observed data. The length vector used to construct this code is obtained from the codeword lengths of C_1. The length vector of C_1 is (0,0,2,8,4,7,1,1,1,1,2). Applying the algorithm of Section 2.2.1 with a short synchronizing codeword 0110, and terminating 0–nodes when possible, the variable length code C_2 is constructed and is listed in Table 6.2:

110	100110*
000	010110*
0110*	001110
0100	101110
0010	100111
1010	010111
1000	001111
1110	1011110
0111	10111110
1111	101111110
00110*	1011111110
10110*	10111111110
10010	10111111111
01010	

Table 6.2: Huffman equivalent code C_2.

To obtain theoretical and observed data, C_2 is substituted for C_1, as depicted in Table 6.4. The simulation model is applied, this time with C_2, producing 100 observed error recovery values. The results are presented graphically on page 132. Observed and theoretical data associated with C_1 and C_2, are given in Table 6.3:

	C_1	C_2
Length vector:	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)
Theoretical Recovery:	3.719	2.424
Observed Average Recovery:	3.697	2.351
Theoretical Variance:	13.043	2.678
Observed Average Variance:	12.524	2.212
Average Codeword Length:	4.122	4.122
Entropy:	4.082	4.082

Table 6.3: Contrasting the robustness of two codes.

Symbol	Probability	C_2
(*space*)	0.190617	110
e	0.103395	000
t	0.0688609	0110*
a	0.0645354	0100
o	0.0622213	0010
i	0.0575601	1010
n	0.055747	1000
s	0.0514721	1110
r	0.0486622	0111
h	0.0471198	1111
d	0.0384853	00110*
l	0.033386	10110*
u	0.0241265	10010
m	0.0228879	01010
c	0.0192597	100110*
w	0.0192339	010110*
y	0.0177917	001110
f	0.0173309	101110
g	0.0155024	100111
p	0.0124589	010111
b	0.0114496	001111
v	0.00782971	1011110
k	0.00618608	10111110
x	0.00131408	101111110
j	0.00125519	1011111110
q	0.000974194	10111111110
z	0.000337817	10111111111

Table 6.4: Code substitution.

Table 6.3 is used to contrast the robustness of C_1 and C_2. To interpret the robustness graphically, observed recovery values are presented together in one graph on page 133.

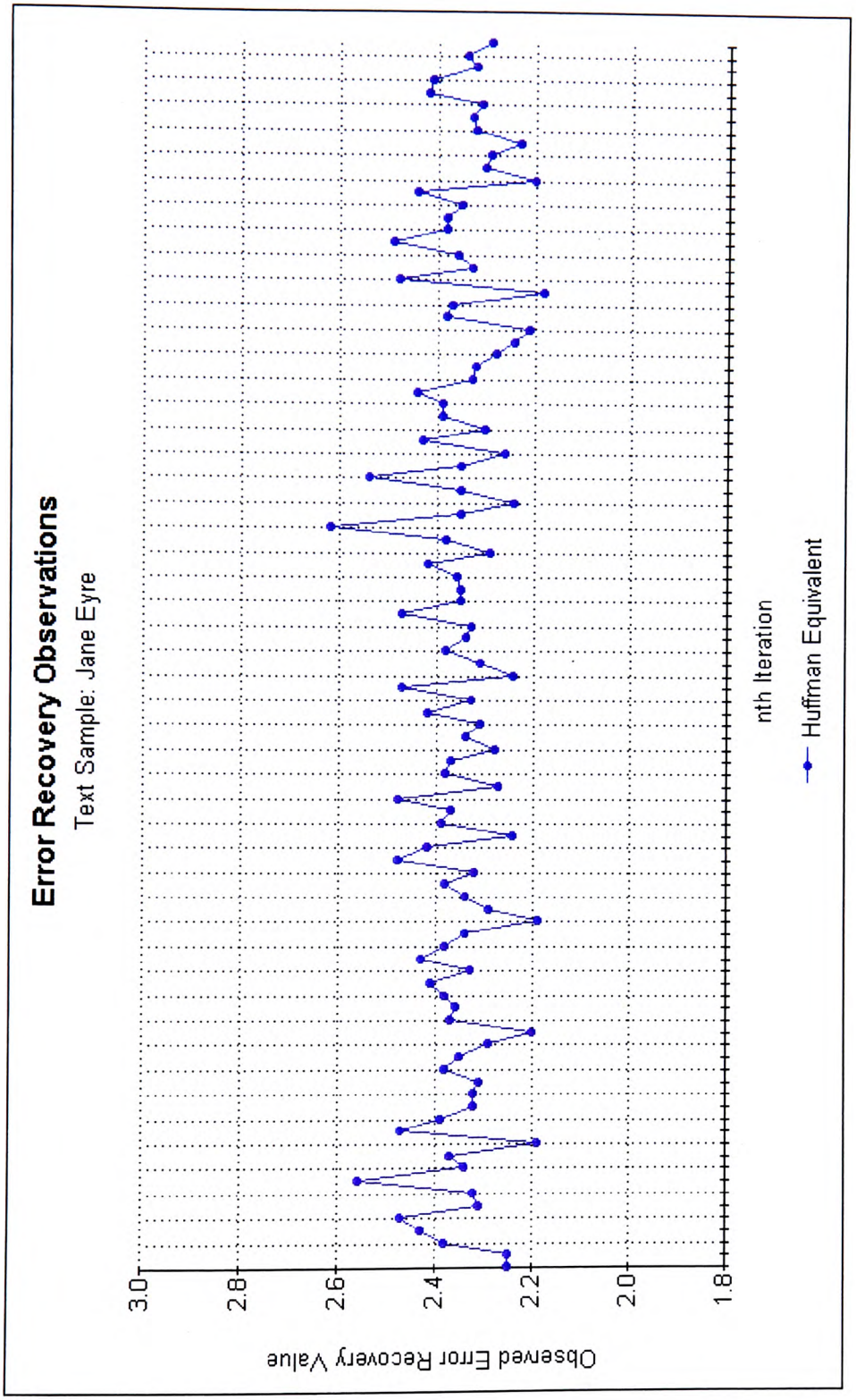


Figure 6.2: Observation results example.

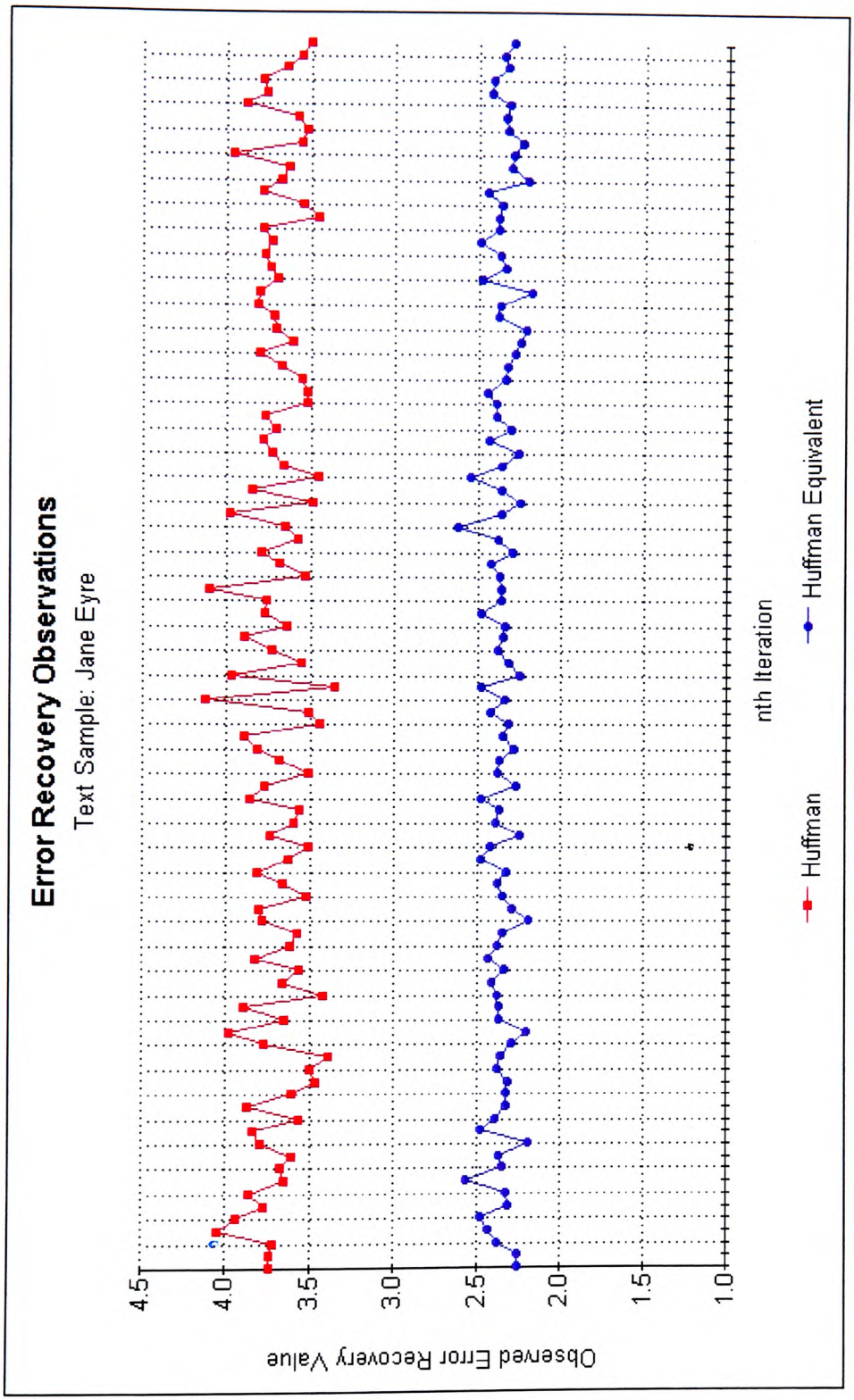


Figure 6.3: Observation results example.

Next we investigate the robustness of T–codes. The T–code C_3 is constructed:

C_3
100
101
1101
1111
0000
0001
0101
1110
0010
0110
00110
00111
01110
01000
011111
011110
010010
110010
010011
110011
110000
1100011
11000101
110001000
1100010010
11000100110
11000100111

Table 6.5: The T–code C_3.

Construction of C_3 involved constructing $C^{(1,3,1,1,1)}_{(0,1,00,01,1100)}$ then applying various node reduction operations to form an optimal code with length vector (0,0,2,8,4,7,1,1,1,1,2). C_3 is substituted for C_2. The simulation model is applied to obtain theoretical data.

In Table 6.6 a compilation of theoretical and observed averages and variances is given. Additionally, all observed error recovery values are presented graphically on page 136.

	C_1	C_2	C_3
Length vector:	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)
Theoretical Recovery:	3.719	2.424	4.841
Observed Average Recovery:	3.697	2.351	4.843
Theoretical Variance:	13.043	2.678	29.294
Observed Average Variance:	12.524	2.212	29.622
Average Codeword Length:	4.122	4.122	4.122
Entropy:	4.082	4.082	4.082

Table 6.6: Contrasting the robustness of three codes.

6.1.1 Code Substitution.

The algorithm defined in Section 2.2.1 will not work for all length vectors. When this occurs a different length vector is used to construct a Huffman equivalent code. For efficiency, the selected vector must produce a code that is optimal or approximately optimal.

For example the text statistics of the sample *The Mill on the Floss*, by George Eliot, produces the Huffman code C with length vector (0,1,0,8,3,9,1,1,1,1,2) and average codeword length 4.128. However the vector (0,1,0,8,3,9,1,1,1,1,2) cannot be used in the algorithm of Section 2.2.1. In this case a different length vector is sought. The vector (0,0,2,8,4,7,1,1,1,1,2) in Section 6.1 was used to construct C_2, thus C_2 is one code that can be used as a substitute. Therefore, substituting C_2 for C, the new average codeword length of C_2 is 4.131. With 4.131 close to the average 4.128, C_2 is considered to be a good choice.

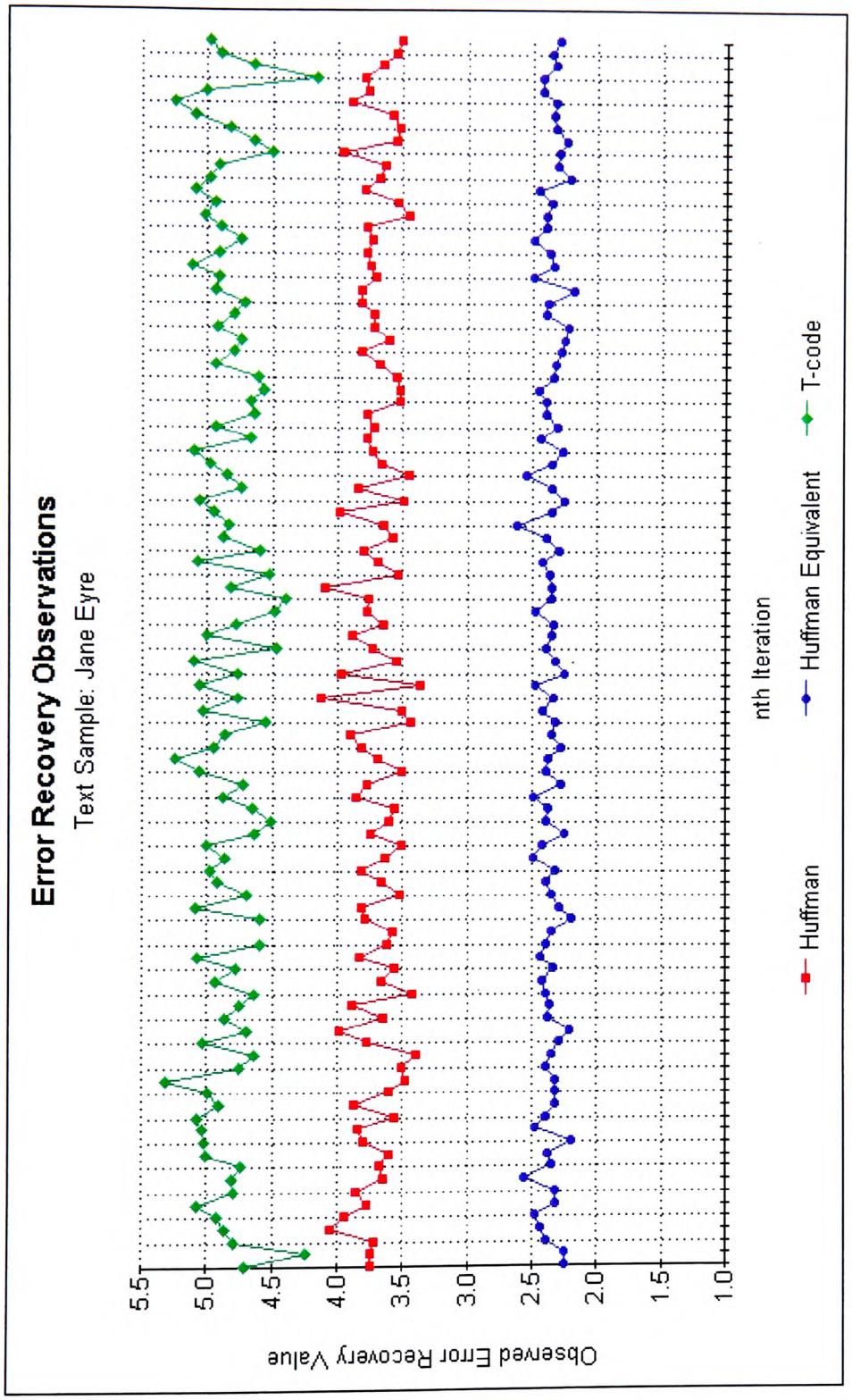


Figure 6.4: Observation results example.

6.2 Robustness Results.

The simulation results are partitioned into two sections. In Section 6.2.1 Huffman equivalent codes that do not require length vector substitution are used. In Section 6.2.2 all Huffman equivalent codes have a substituted length vector. The robustness results in Section 6.2.1 and Section 6.2.2 are specific to optimal and suboptimal Huffman equivalent codes, respectively. In both sections all T-codes are optimal, each one was made optimal by the application of node reduction and/or extension operations.

6.2.1 Robustness of Variable Length Codes.

1. Text sample *Mansfield Park* by Jane Austen. Error recovery observations are given on page 139:

	Huffman	Huffman Equivalent	T-code
Length Vector:	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)
Theoretical Recovery:	4.182	2.420	4.835
Observed Recovery Average:	4.160	2.305	4.807
Theoretical Variance:	14.377	2.741	29.393
Observed Variance Average:	13.572	1.980	28.609
Average Codeword Length:	4.126	4.126	4.126
Entropy:	4.088	4.088	4.088

2. Text sample *War of the Worlds* by H. G. Wells. Error recovery observations are given on page 140:

	Huffman	Huffman Equivalent	T-code
Length Vector:	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)
Theoretical Recovery:	4.455	2.396	4.743
Observed Recovery Average:	3.996	2.358	4.431
Theoretical Variance:	19.243	2.662	28.200
Observed Variance Average:	13.343	2.362	22.245
Average Codeword Length:	4.123	4.123	4.123
Entropy:	4.080	4.080	4.080

3. Text sample *Wuthering Heights* by Emily Bronte. Error recovery observations are given on page 141:

	Huffman	Huffman Equivalent	T–code
Length Vector:	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)
Theoretical Recovery:	4.820	2.421	4.820
Observed Recovery Average:	4.348	2.338	4.561
Theoretical Variance:	23.086	2.673	29.086
Observed Variance Average:	15.642	2.169	24.092
Average Codeword Length:	4.115	4.115	4.115
Entropy:	4.075	4.075	4.075

4. Text sample *The Time Machine* by H. G. Wells. Error recovery observations are given on page 142:

	Huffman	Huffman Equivalent	T–code
Length Vector:	(0,0,1,1,5,5,9,1,1,1,1,2)	(0,0,1,1,5,5,9,1,1,1,1,2)	(0,0,1,1,5,5,9,1,1,1,1,2)
Theoretical Recovery:	5.727	1.829	4.275
Observed Recovery Average:	5.389	1.828	4.123
Theoretical Variance:	30.651	0.940	20.722
Observed Variance Average:	27.187	0.985	18.515
Average Codeword Length:	4.126	4.126	4.126
Entropy:	4.083	4.083	4.083

5. Text sample *Paradise Lost* by John Milton. Error recovery observations are given on page 143:

	Huffman	Huffman Equivalent	T–code
Length Vector:	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)
Theoretical Recovery:	4.033	2.412	4.762
Observed Recovery Average:	3.890	2.324	5.049
Theoretical Variance:	13.281	2.705	28.192
Observed Variance Average:	11.894	2.060	34.620
Average Codeword Length:	4.122	4.122	4.122
Entropy:	4.088	4.088	4.088

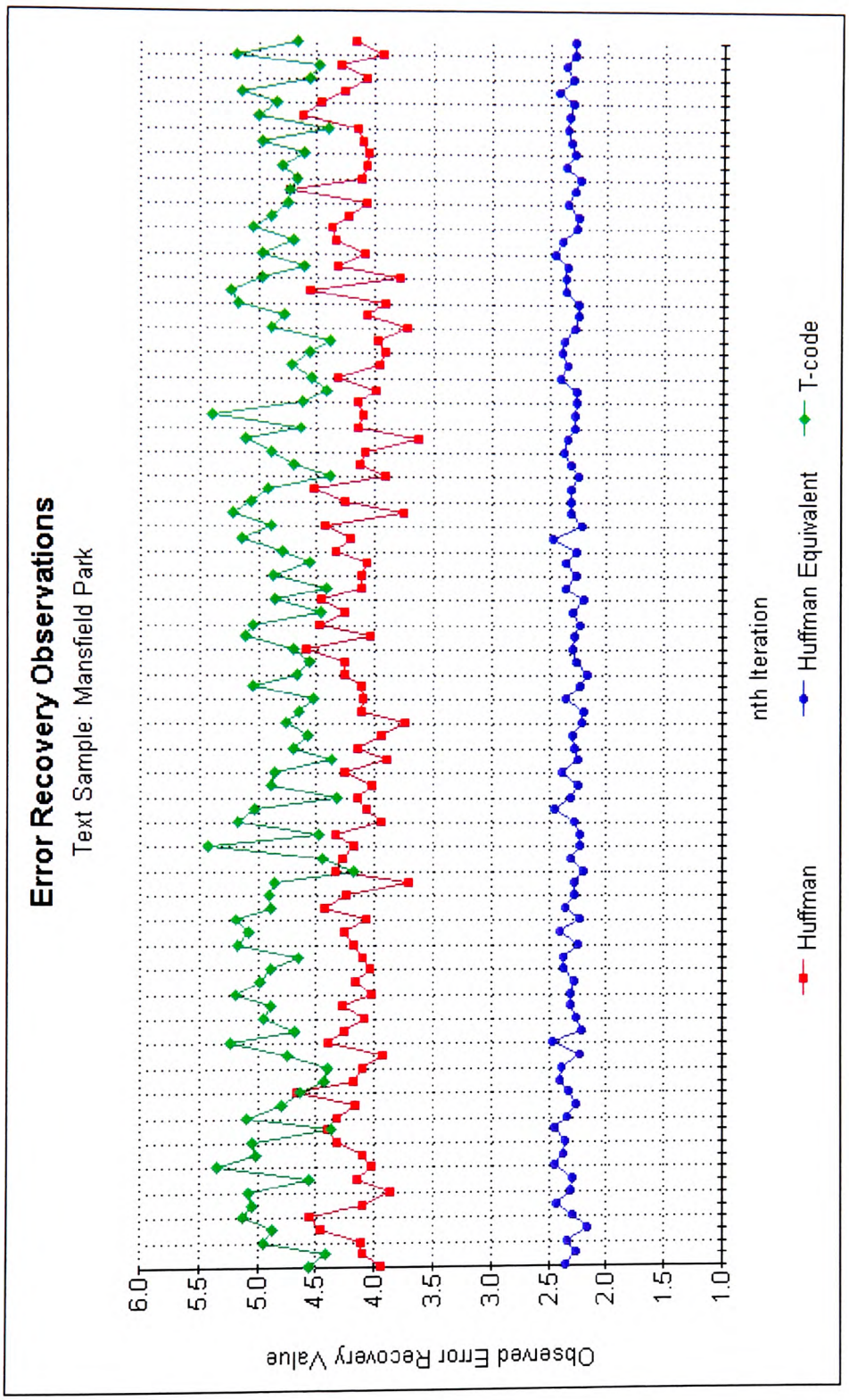


Figure 6.5: Observation results (1.)

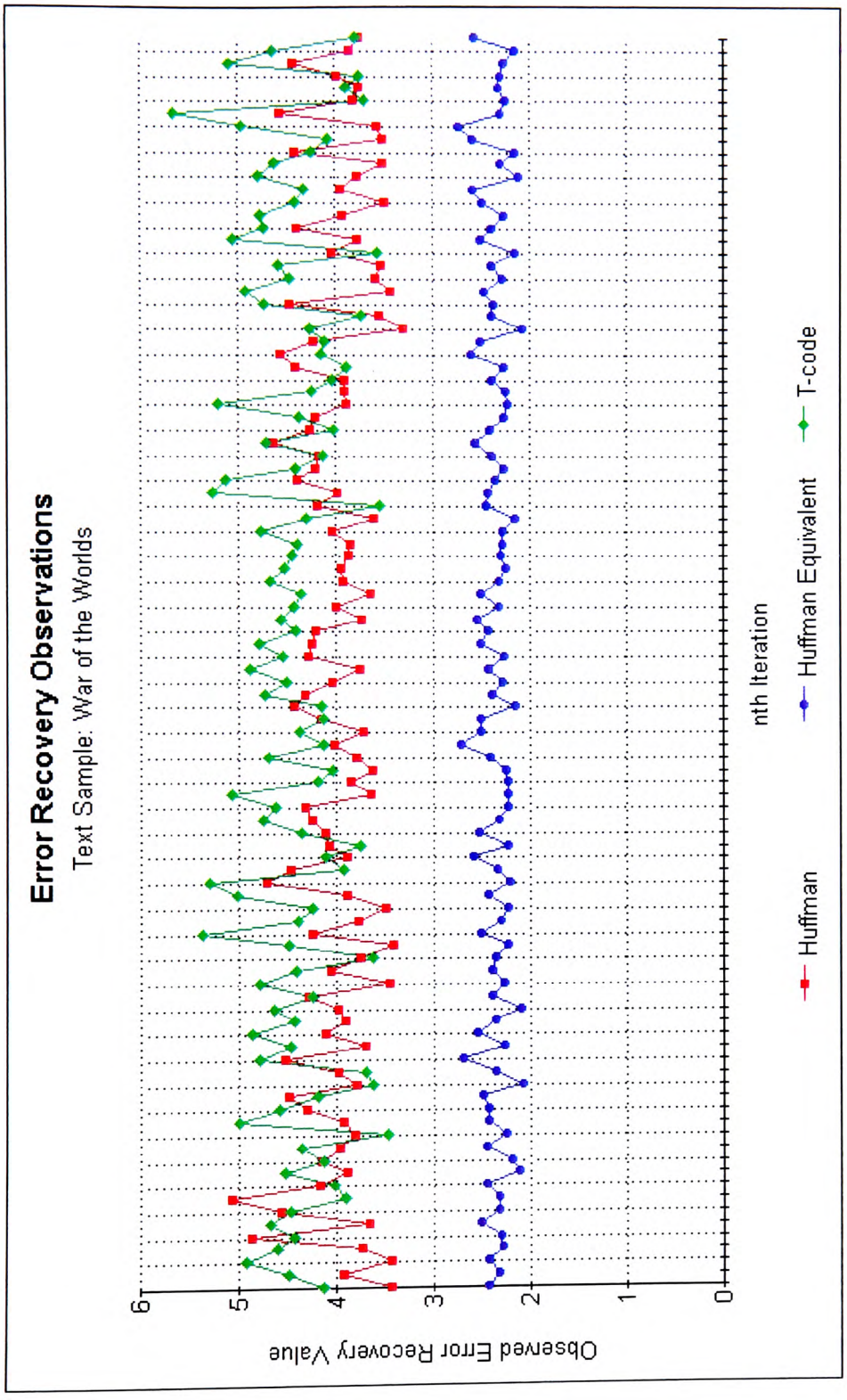


Figure 6.6: Observation results (2.)

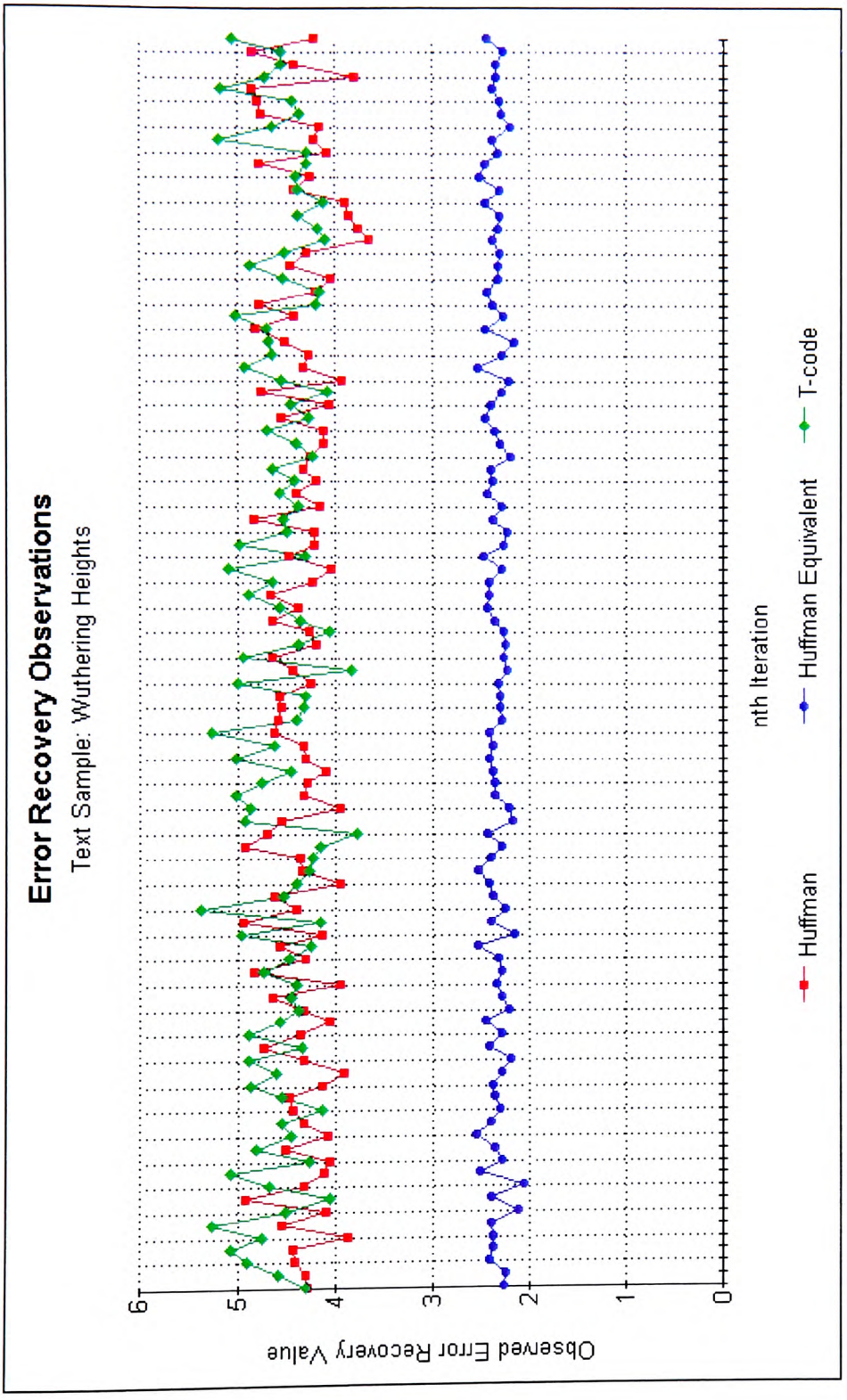


Figure 6.7: Observation results (3.)

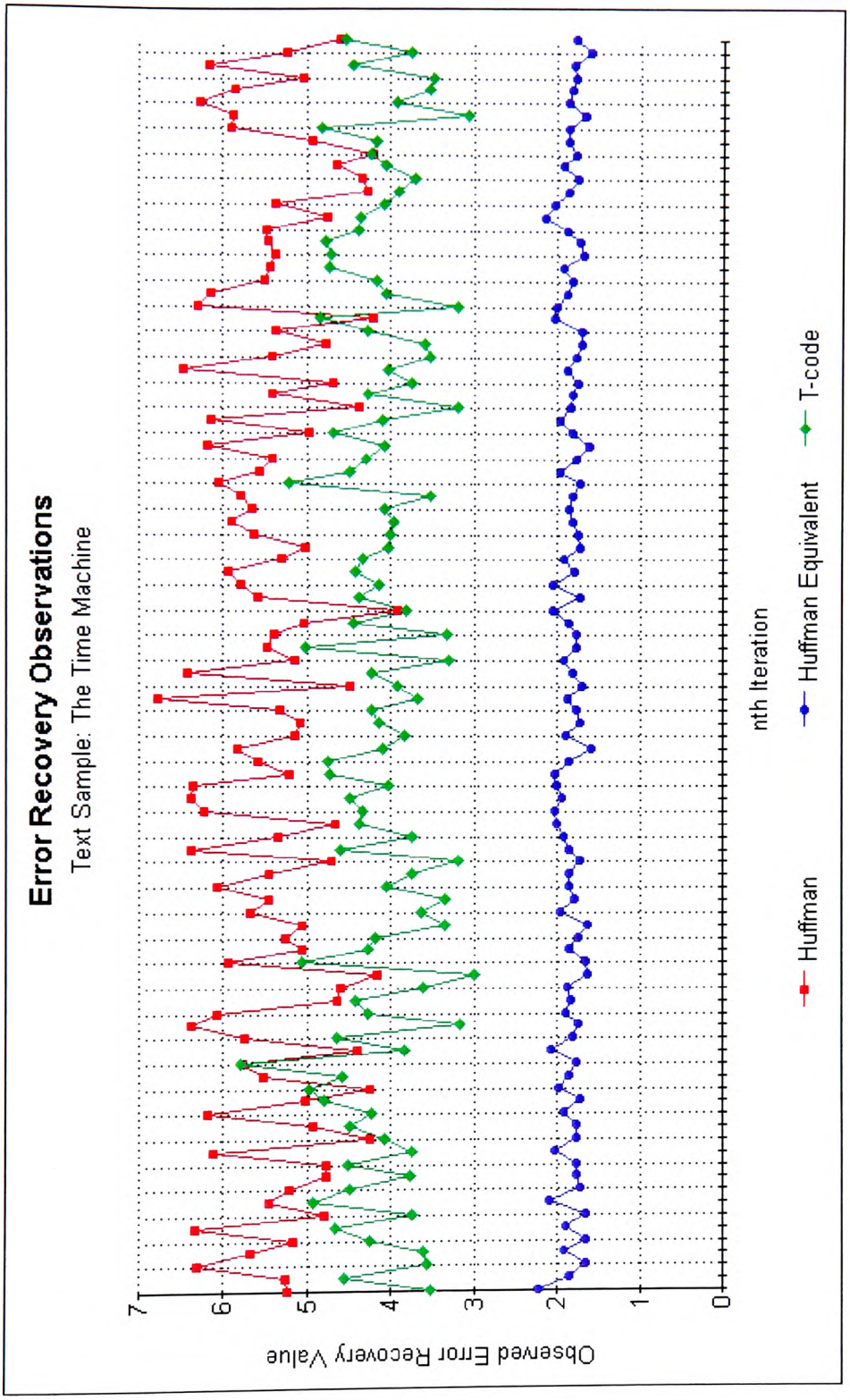


Figure 6.8: Observation results (4.)

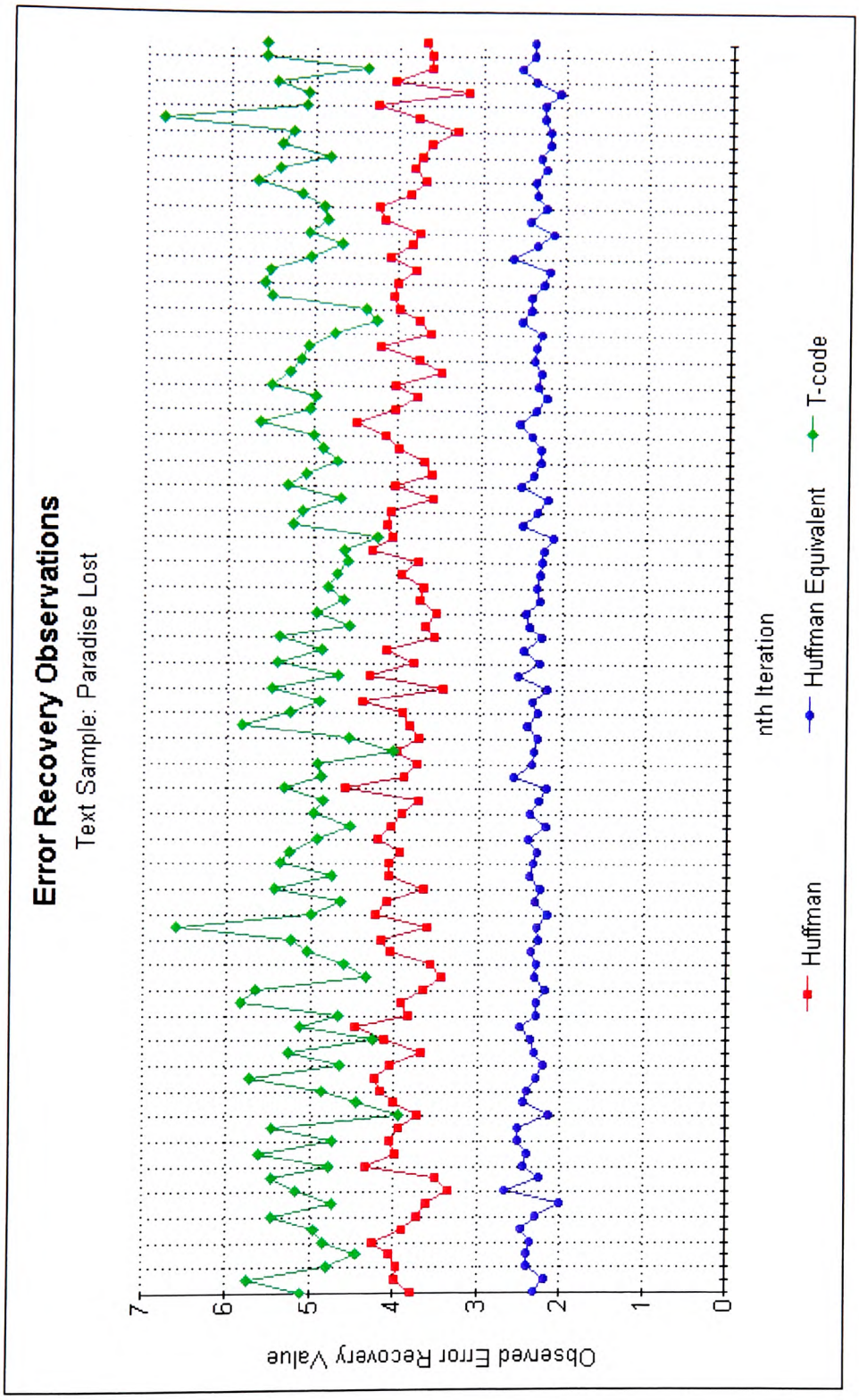


Figure 6.9: Observation results (5.)

6.2.2 Robustness of Variable Length Codes with Length Vector Substitution.

1. Text sample *Crime and Punishment* by Fyodor Dostoyevsky. Error recovery observations are given on page 145:

	Huffman	Huffman Equivalent	T-code
Length Vector:	(0,1,0,8,4,6,3,1,0,4)	(0,0,2,8,4,7,1,1,1,1,2)	(0,1,0,8,4,6,3,1,0,4)
Theoretical Recovery:	4.671	2.410	4.236
Observed Recovery Average:	4.783	2.338	4.318
Theoretical Variance:	26.536	2.671	31.722
Observed Variance Average:	26.831	2.254	34.135
Average Codeword Length:	4.125	4.136	4.125
Entropy:	4.085	4.085	4.085

2. Text sample *Mayor of Casterbridge* by Thomas Hardy. Error recovery observations are given on page 146:

	Huffman	Huffman Equivalent	T-code
Length Vector:	(0,0,2,8,4,7,1,1,0,4)	(0,0,2,8,4,7,1,1,1,1,2)	(0,0,2,8,4,7,1,1,0,4)
Theoretical Recovery:	5.172	2.420	3.383
Observed Recovery Average:	4.885	2.404	3.735
Theoretical Variance:	28.281	2.715	12.517
Observed Variance Average:	21.731	2.531	16.567
Average Codeword Length:	4.1196	4.1202	4.1196
Entropy:	4.081	4.081	4.081

3. Text sample *Adam Bede* by George Eliot. Error recovery observations are given on page 147:

	Huffman	Huffman Equivalent	T-code
Length Vector:	(0,1,0,8,3,9,1,1,1,1,2)	(0,0,2,8,4,7,1,1,1,1,2)	(0,1,0,8,3,9,1,1,1,1,2)
Theoretical Recovery:	5.349	2.405	3.392
Observed Recovery Average:	5.561	2.298	3.272
Theoretical Variance:	40.354	2.663	14.972
Observed Variance Average:	42.600	2.011	12.745
Average Codeword Length:	4.107	4.113	4.107
Entropy:	4.066	4.066	4.066

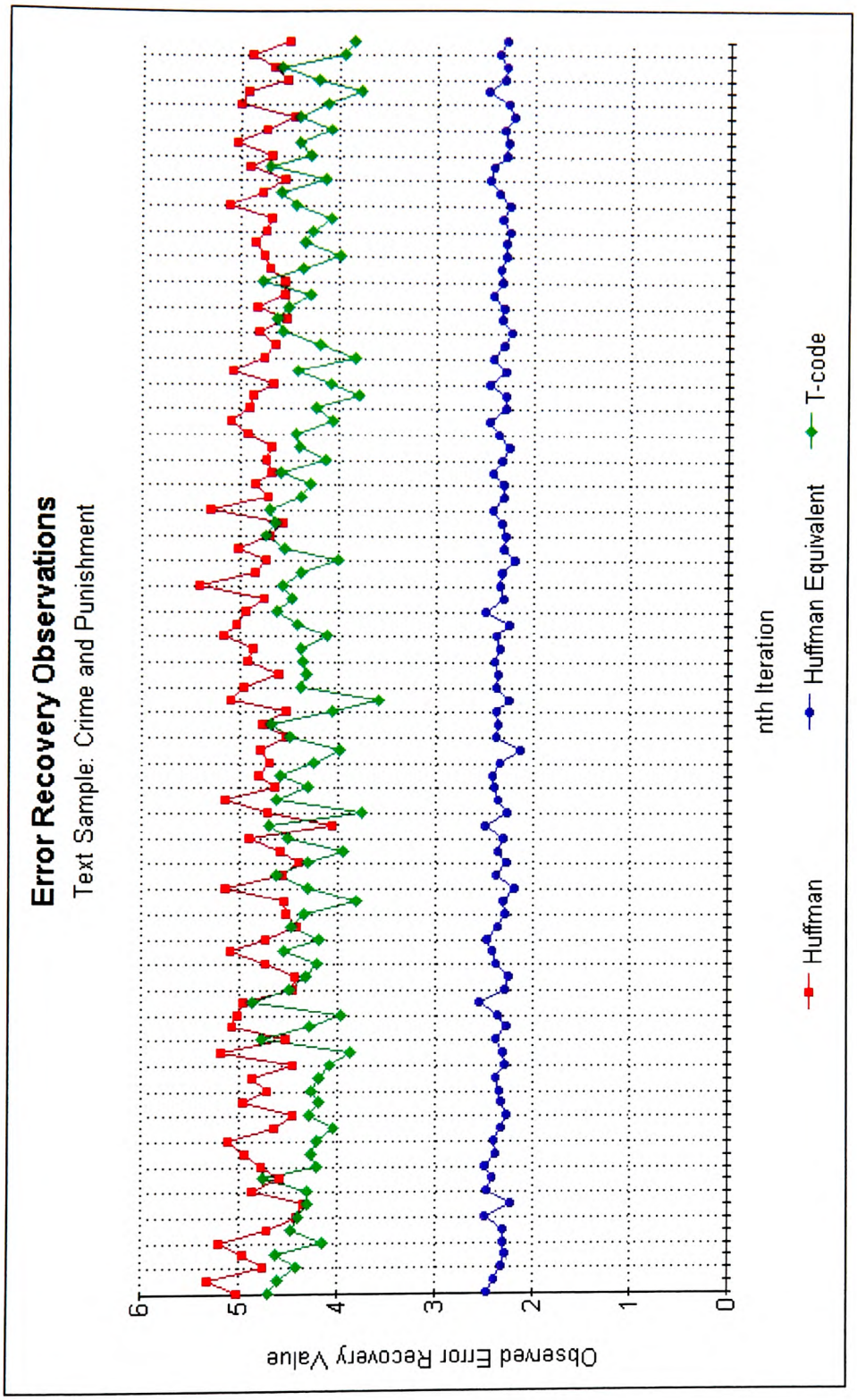


Figure 6.10: Observation results (1.)

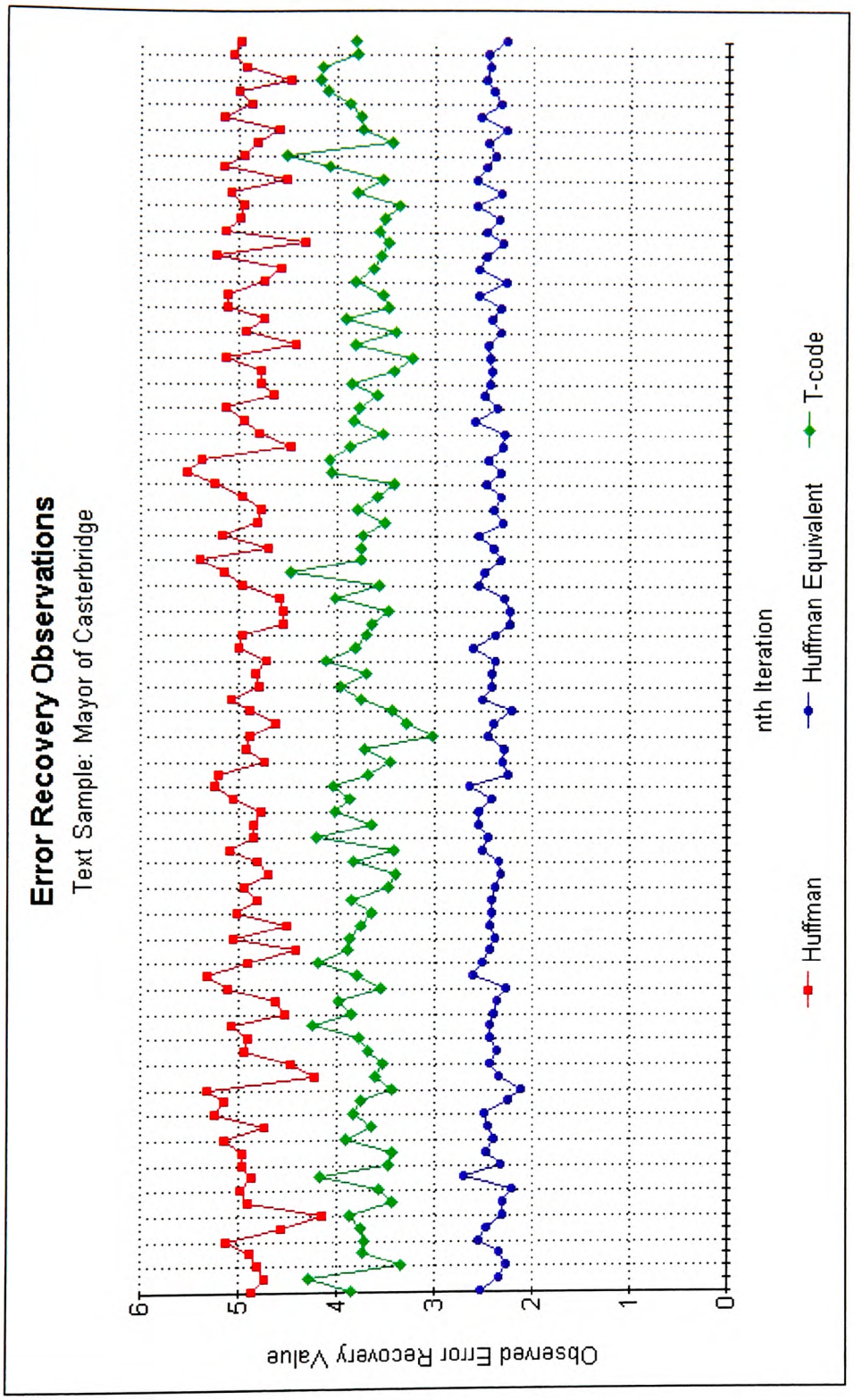


Figure 6.11: Observation results (2.)

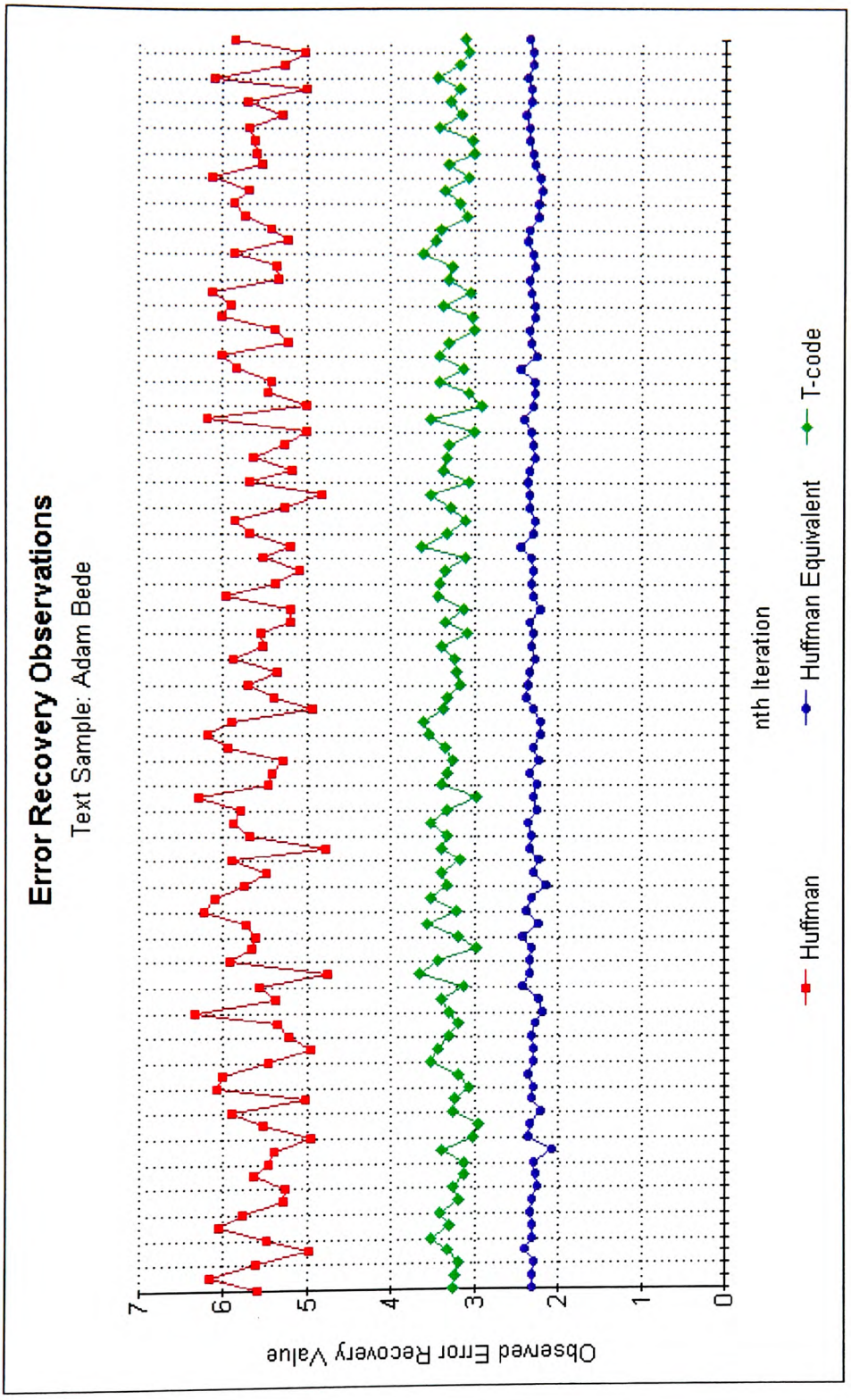


Figure 6.12: Observation results (3.)

All codes presented in this section lead to the conclusion that Huffman equivalent codes are more robust than either Huffman codes or T–codes with node reduction/extension.

6.3 Further Results.

In this section we observe the error recovery of T–codes that are not node reduced or node extended. The following two examples will illustrate that such T–codes, when they can be constructed, are competitive. In both examples the number of codewords in each T–code is kept as close as possble to 27 (the number of codewords used in Sections 6.2.1 and 6.2.2).

Example 1. Two T–codes $C^{(1,1,1,1,1)}_{(0,1,00,01,11)}$ and $C^{(3,1,1,1)}_{(0,1,01,11)}$ are constructed, each consisting of 33 codewords and with the same length vector (0,0,2,6,6,8,6,3,2). Both codes are tabulated in Table 6.8. For this example it was possible to construct a simple and a generalised T–code, both were constructed from the starting set $C = \{0, 1\}$ without applying node reduction/extension operations.

The novel *Pride and Prejudice*, by Jane Austen, is used as a text sample. Before applying the simulation model, 33 characters are selected. The transmisson probability of each character is determined then used to construct a Huffman code C_1 with length vector (0,0,2,8,2,10,2,2,2,3,2). C_1 is also tabulated in Table 6.8 along with the Huffman equivalent code C_2, constructed from the same length vector.

The simulation model is applied, producing the observed error recovery values that are depicted graphically on page 150. Calculating observed averages and theoretical values, Table 6.7 is formed. In this case, the error recovery results imply that the generalised T–code is more robust. In comparison, the Huffman equivalent code is not as robust. It must be noted however that the T–codes have greater average codeword length.

	C_1	C_2	$C^{(1,1,1,1,1)}_{(0,1,00,01,11)}$	$C^{(3,1,1,1)}_{(0,1,01,11)}$
Theoretical Recovery:	3.804	2.387	2.074	1.881
Observed Recovery Average:	3.639	2.294	2.088	1.857
Theoretical Variance:	13.478	2.763	1.362	0.805
Observed Variance Average:	10.238	2.112	1.430	0.729
Average Codeword Length:	4.243	4.243	4.277	4.277
Entropy:	4.209	4.209	4.209	4.209

Table 6.7: Observations of Example 1.

Symbol	Transmission Probability	C_1	C_2	$C^{(1,1,1,1,1)}_{(0,1,00,01,11)}$	$C^{(3,1,1,1)}_{(0,1,01,11)}$
'space'	0.179576	111	110	100	001
e	0.102727	010	000	101	101
t	0.0691742	1100	0110*	0000	0001
a	0.0618833	1001	0100	0001	0000
o	0.0588227	1000	0010	0011	1001
n	0.0562899	0111	1010	0101	0101
i	0.0560913	0110	1000	0111	0111
h	0.0503711	0010	1110	1111	1111
s	0.049113	0001	0111	00100	10001
r	0.0480995	0000	1111	00101	10000
d	0.0330872	10110	00110*	01100	01001
l	0.0319781	10100	10110*	01101	01101
m	0.0218343	110111	100110*	11100	11001
u	0.021746	110110	010110*	11101	11101
c	0.0203666	110101	100100	010000	010001
y	0.0184611	110100	010100	010001	010000
f	0.0180141	101111	001110	010011	011001
w	0.0179405	101110	101110	110000	110001
g	0.0149572	101010	100111	110001	110000
b	0.0133717	001111	010111	110011	111001
,	0.0132834	001110	001111	110101	110101
p	0.0121946	001100	101111	110111	110111
.	0.00893348	1010111	1001010	0100100	0110001
v	0.00847366	1010110	0101010	0100101	0110000
k	0.00466263	00110111	10010110*	1100100	1110001
;	0.00222923	00110100	01010110*	1100101	1110000
j	0.0013243	001101011	100101110	1101100	1101001
z	0.0013059	001101010	010101110	1101101	1101101
x	0.00125072	0011011011	1001011110	11010000	11010001
q	0.000936204	0011011010	0101011110	11010001	11010000
!	0.000669505	0011011000	1001011111	11010011	11011001
?	0.000623523	00110110011	01010111110	110100100	110110001
:	0.000207841	00110110010	01010111111	110100101	110110000

Table 6.8: Variable length codes.

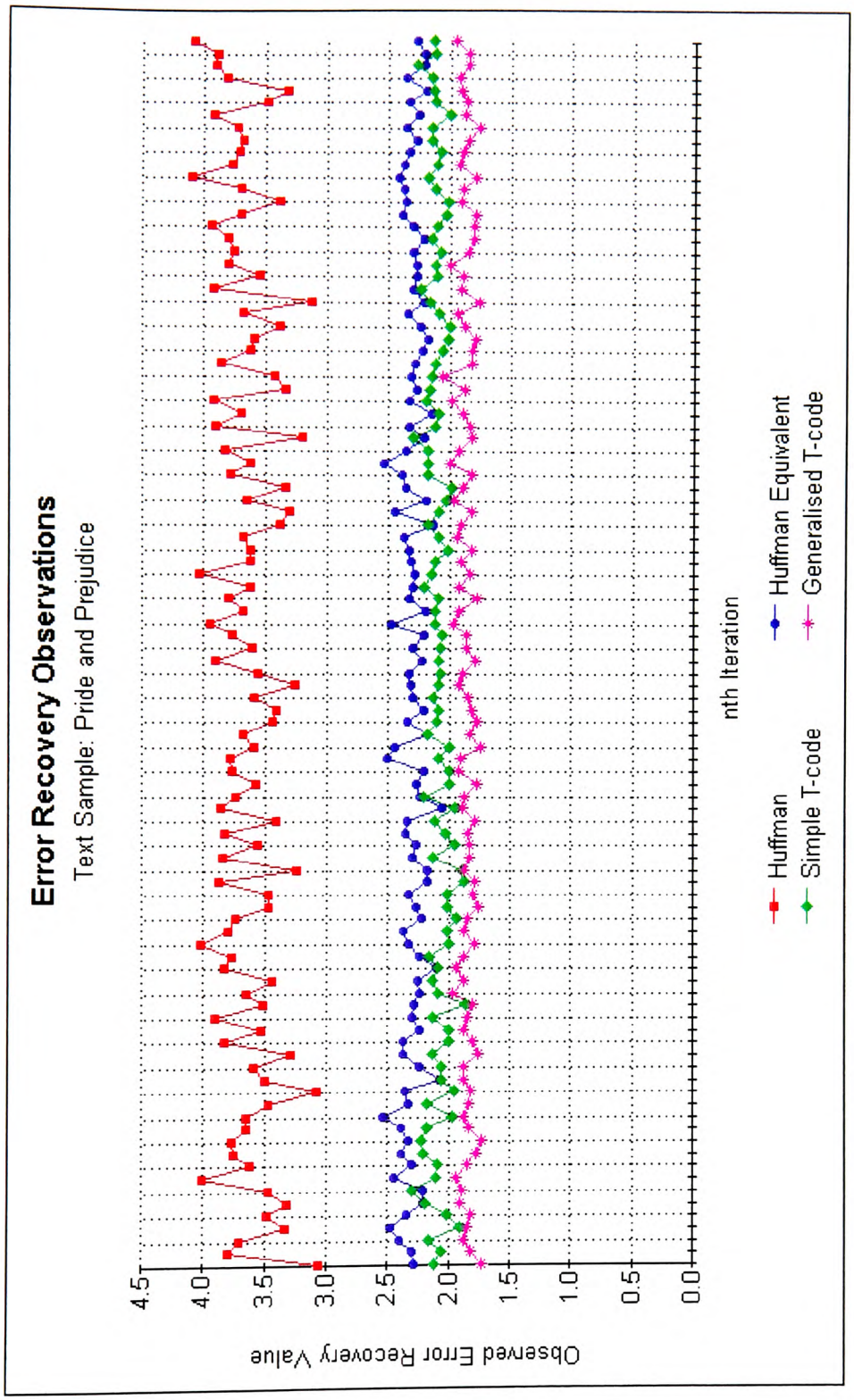


Figure 6.13: Observations for Example 1.

Example 2. The T–code $C^{(4,1,2)}_{(0,1,01)}$ is constructed from the starting set $C = \{0, 1\}$ without applying node reduction/extension operations. Table 6.10 lists the code $C^{(4,1,2)}_{(0,1,01)}$, it consists of 31 codewords with a length vector (0,1,2,3,5,6,5,4,3,2). A simple T–code consisting of 31 codewords and constructed without applying node reduction/extension operations, could not be found.

The novel *Dubliners*, by James Joyce, is used as a text sample. Before applying the simulation model, 31 characters are selected. The transmisson probability of each character is determined then used to construct a Huffman code C_1 with length vector (0,0,2,8,2,11,1,1,0,2,4).

The algorithm given in Section 2.2.1 cannot be used to construct a Huffman equivalent code from the length vector (0,0,2,8,2,11,1,1,0,2,4). Instead, the length vector (0,1,2,3,5,6,5,4,3,2) for the T–code $C^{(4,1,2)}_{(0,1,01)}$ is used to construct two Huffman equivalent codes: C_2 and C_3, that are also given in Table 6.10. In the construction of C_2 the short synchronizing codeword 010 was chosen and 1–nodes were terminated whenever possible. C_3 was constructed using the short synchronizing codeword 011 and terminating 0–nodes.

The observed error recovery values of C_1, C_2 and $C^{(4,1,2)}_{(0,1,01)}$ are graphically presented on page 153. Similarly, the graph on page 154 illustrates the observed error recovery values of C_1, C_3, and $C^{(4,1,2)}_{(0,1,01)}$.

Calculating observed averages and theoretical values, Table 6.9 is formed:

	C_1	C_2	C_3	$C^{(4,1,2)}_{(0,1,01)}$
Theoretical Recovery:	3.176	1.721	1.987	1.851
Observed Recovery Average:	3.018	1.688	1.936	1.827
Theoretical Variance:	6.272	0.625	0.782	0.692
Observed Variance Average:	5.095	0.556	0.592	0.604
Average Codeword Length:	4.215	4.286	4.286	4.286
Entropy:	4.174	4.174	4.174	4.174

Table 6.9: Observations of Example 2.

In this example the Huffman equivalent code C_2 appears to be more robust than the T–code $C^{(4,1,2)}_{(0,1,01)}$.

Symbol	Transmission Probability	C_1	C_2	C_3	$C^{(4,1,2)}_{(0,1,01)}$
'space'	0.186882	111	10	11	11
e	0.098621	001	010*	011*	001
t	0.0677727	1011	011	000	101
a	0.065704	1001	0010*	0011*	0001
o	0.0586169	1000	0011	1011*	1001
h	0.0552445	0111	1111	1000	0111
n	0.0544712	0110	00010*	10011*	00001
i	0.0518998	0101	11010*	01011*	00000
s	0.049624	0100	00000	10100	10001
r	0.0451938	0000	00011	01000	01001
d	0.037684	11001	11011	00100	01101
l	0.0336212	10101	000010*	010011*	100001
u	0.0208389	110111	110010*	001011*	100000
m	0.0206041	110110	111010*	101011*	010001
w	0.0199081	110101	000011	100100	011001
g	0.0179472	110001	110011	010100	010111
c	0.017845	110000	111011	010010	010101
f	0.0171269	101001	1100010*	1001011*	0100001
y	0.0155387	101000	1110010*	0101011*	0100000
.	0.0121692	000111	1100000	1001010	0110001
p	0.0120228	000110	1100011	0010100	0101001
b	0.0115367	000101	1110011	1010100	0101101
:	0.0108766	000100	11000010*	00101011*	01100001
k	0.00773622	1101000	11100010*	10101011*	01100000
v	0.00604315	11010011	11000011	01010100	01010001
j	0.00120697	1101001010	11100011	00101010	01011001
!	0.000817537	1101001000	111000010*	010101011*	010100001
x	0.000809251	11010010111	111000000	010101010	010100000
q	0.000626963	11010010110	111000011	101010100	010110001
z	0.000527532	11010010011	1110000010*	1010101011*	0101100001
;	0.000483341	11010010010	1110000011	1010101010	0101100000

Table 6.10: Variable length codewords.

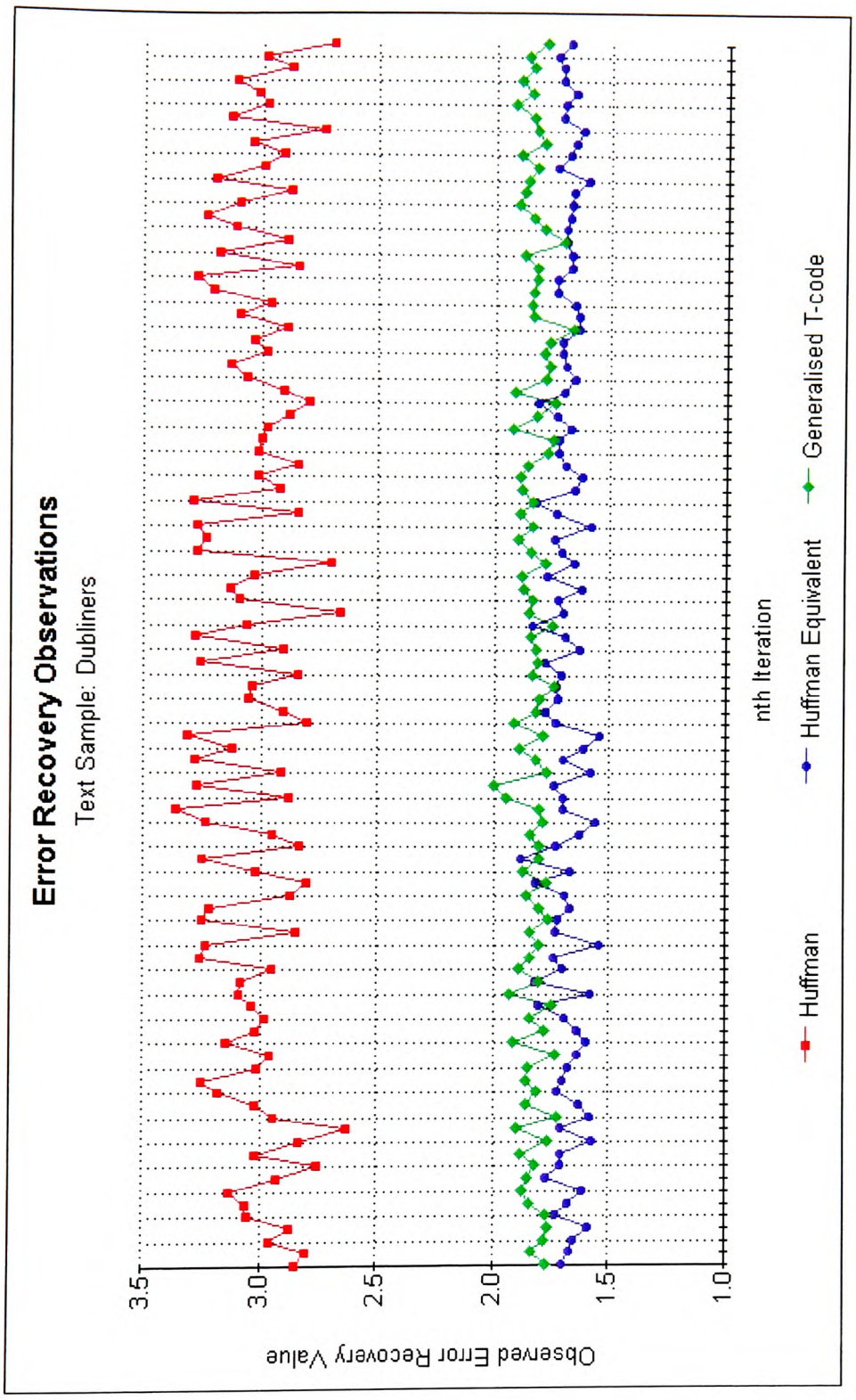


Figure 6.14: Observations for Example 2. Codes C_1, C_2, and $C_{(0,1,01)}^{(4,1,2)}$.

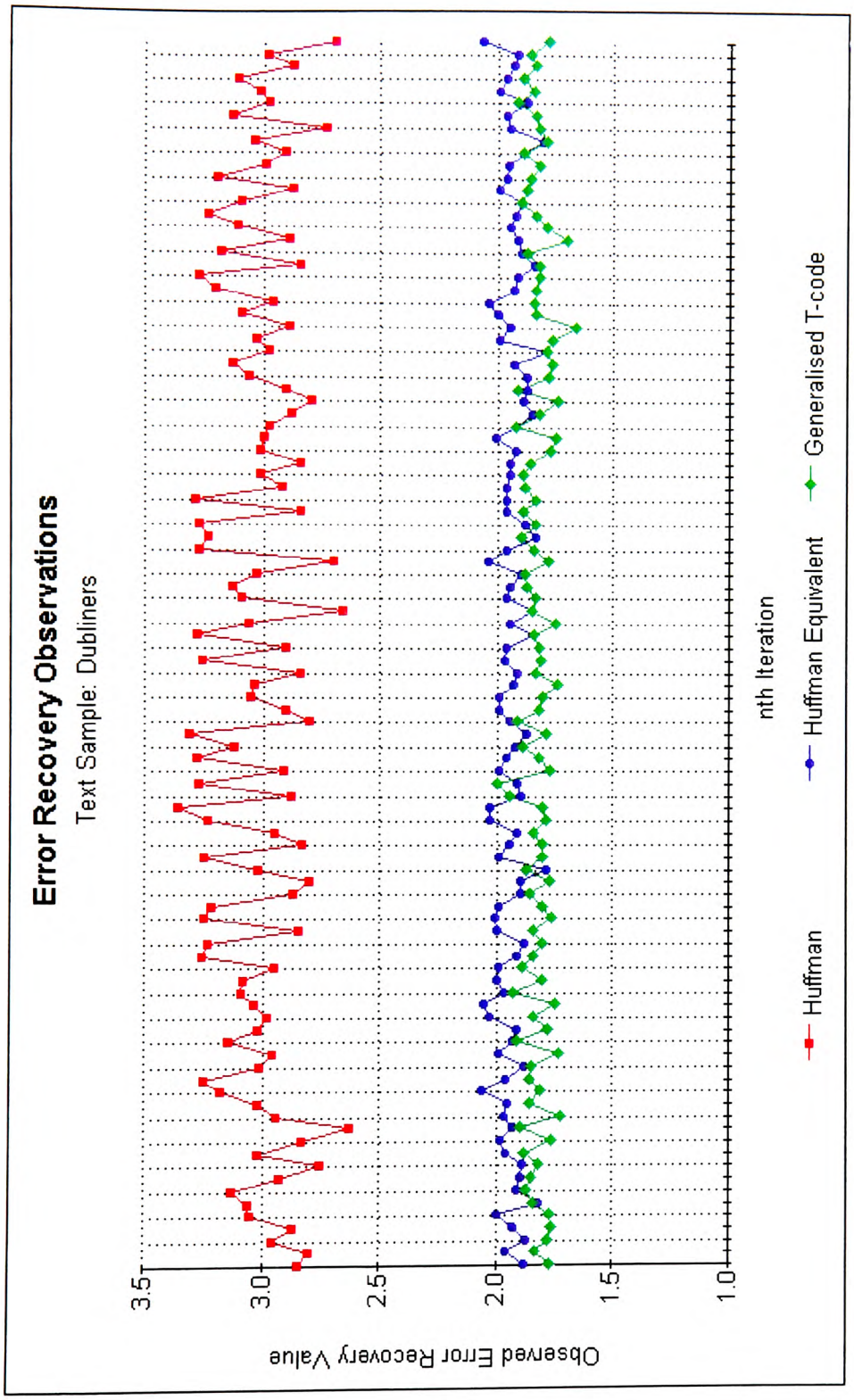


Figure 6.15: Observations for Example 2. Codes C_1, C_3, and $C^{(4,1,2)}_{(0,1,01)}$.

Chapter 7

Conclusions.

The observed error recovery results in Sections 6.2.1, 6.2.2 and 6.3 demonstrate that Huffman equivalent codes are more robust than Huffman codes. In Sections 6.2.1 and 6.2.2 T–codes were found to be less robust than Huffman equivalent codes, indeed often less robust than Huffman codes. However, in Section 6.3 it is shown that T–codes may be more robust than Huffman equivalent codes if T–codes can be constructed without node reductions or extensions.

It should also be noted that it is not always possible to construct Huffman equivalent codes with the ideal length vector. A suboptimal code may be necessary. The results in Section 6.2.2 demonstrated that a suitable length vector was able to be found and used to construct a robust code from the Huffman equivalent algorithm with a marginal reduction in optimality.

The robustness of T–codes varies: some T–codes are competitive with Huffman codes and even with Huffman equivalent codes, yet other T–codes are much less competitive. This disparity may be understood if we consider how the T–codes were constructed. In Sections 6.2.1 and 6.2.2 each T–code was constructed from the T–code algorithm defined in Section 2.3.1 (simple T–codes) or Section 2.3.4 (generalised T–codes), then altered by node reduction and/or extension operations to form a modified T–code. In Section 6.3 T–codes were constructed without node reduction/extension operations. The synchronization properties of modified T–codes were affected by the node reduction/extension operations. Node reduction and extension operations can be seen to reduce the synchronization properties of T–codes.

In terms of robustness, an unmodified T–code may be preferred if node reduction/extension operations have adverse affects on synchronization. However, constructing such a code for any length

vector may be difficult, if not impossible. For instance, for all length vectors in Section 6.2.1 and 6.2.2, it remains unknown if any unmodified T–codes can be constructed. It was therefore necessary to use T–codes that were constructed from node reduction and/or extension operations. Thus the favourable properties of unmodified T–codes may not be applicable in practice. The same may be said to some Huffman equivalent codes, however it appears to be generally possible to work with a satisfactory substitute length vector.

7.1 Relationship Between Variance and Error Recovery Values.

In the results presented, variances tend to be larger when the error recovery value is larger. Thus a robust code will minimise data corruption with some regularity.

7.2 Observed and Theoretical Measures.

Comparing the observed and theoretical measures in Chapter 6, the theoretical error recovery E_{rec} was approximately equal to the observed error recovery average.

The theoretical variance, derived in Section 4.3.5, was not always close to the average observed variance. However, the theoretical variance was close to the observed variance when the code considered was robust. However, these differences may appear less significant when we consider the standard deviation (instead of the variance).

As the theoretical measures of error recovery and variance are approximately the same as the observed measures, they may be used confidently in a theoretical framework.

7.3 Further Work.

This work suggests some further area of research. It would be useful to know when a substitute length vector for a Huffman equivalent code will be necessary and how that length vector should be formed. Another important area is to determine when unmodified optimal T–codes exist and, if possible, how node reduction/extension operations should be undertaken to preserve synchronization properties of T–codes.

Bibliography

[1] I. Anderson; *A First Course in Combinatorial Mathematics*; Oxford Applied Mathematics and Computing Science Series. 1996, pp. 67–78.

[2] *Blackmask Online*; http://www.blackmaskonline.com.

[3] A.E. Escott and S. Perkins; *Binary Huffman Equivalent Codes with a Short Synchronizing Codeword*; IEEE Transactions on Information Theory. Vol. 44, No. 1, Jan 1998, pp. 346–351.

[4] T.J. Ferguson and J.H. Rabinowitz; *Self-Synchronizing Huffman Codes*; IEEE Transactions on Information Theory. Vol. IT–30, No. 4, Jul 1984, pp.687–693.

[5] W.F. Friedman; *Elements of Cryptanalysis*; Laguna Hills, CA: Aegean Park, 1976.

[6] G.R. Grimmett and D.R. Stirzaker; *Probability and Random Processes*; Oxford Science Publications. pp. 117. 1992.

[7] R. W. Hamming; *Coding and Information Theory*; Prentice–Hall. 1980.

[8] D.A. Huffman; *A method for the construction of minimum redundancy codes*; Proc. IRE. Vol. 40, No. 2, 1952, pp. 1098–1101.

[9] S.J. Mason; *Feedback Theory - Further Properties of Signal Flow Graphs*; Proc. Inst. Radio Eng. Vol. 44, July 1956, pp. 920–926.

[10] J.C. Maxted and J.P. Robinson; *Error Recovery for Variable Length Codes*; IEEE Transactions on Information Theory. Vol. IT–31. No. 6, Nov 1985, pp. 794–801.

[11] M. Nelson; *The Data Compression Book*; Prentice Hall and M&T Books. 1991.

[12] S. Perkins and A.E. Escott; *Synchronizing Codewords of q-ary Huffman Codes*; Discrete Maths. Vol. 197/198, 1999, pp. 637-655.

[13] Y. Takishima, M. Wada and H. Murakami; *Error States and Synchronization Recovery for Variable Length Codes*; IEEE Transactions on Communications. Vol. 42, No. 2/3/4, Feb/Mar/Apr 1994, pp. 783–792.

[14] K. Thomas; *Entropy - The Key to Data Compression*; Dr Dobbs Journal. February 1991, pp. 32–34.

[15] M.R. Titchener; *Generalised T-codes: extended construction algorithm for self-synchronizing codes*; IEE Proc.– Commun. Vol. 143, No. 3, Jun. 1996, pp. 122–128.

[16] M.R. Titchener; *The Synchronization of Variable Length Codes*; IEEE Transactions on Information Theory. Vol. 43, No. 2, Mar. 1997, pp. 683–691.

[17] D. Welsh; *Codes and Cryptography*; Oxford Science Publications. 1988.

[18] R.J. Wilson; *Introduction to Graph Theory*; Longman. 1996, pp. 109–110.